Unicenter

Management Services
Network Control Language
User's Guide

8th edition

P01- 030

Computer Associates
The Software That Manages eBusiness

Booklist

Edition	Publication Number	Product Version	Min. MS Level	Publish Date
8th Edition	P01-030	Version 4.1	MS 4.1	December 2000

Table of Contents

Chapter 6 Designing Interactive Panels (Panel Services) 6-1

Figures and Tables

What's New in This Edition

This manual is the 8th Edition of the *Network Control Language User's Guide*, publication number P01-030.

Since the last release of Management Services, this manual has been restructured to simplify its use and to remove references to NCL/EF, which has been superseded by Netmaster Databases (NDB). This restructure involved the following changes:

- Chapter 16, *Introduction to NCL/EF*, has been removed, and all following chapters have been renumbered accordingly.
- Appendix A, *Customer Support Services*, has been removed, and all following appendixes have been relettered accordingly.
- Information within various chapters has been restructured as necessary.

About This Guide

This guide and the *Network Control Language Reference* are the principal reference documents for users of the Network Control Language (NCL).

This guide provides information for anyone who writes or maintains NCL procedures. It describes the principles behind NCL and includes reference material and examples. Early chapters discuss the data management and file processing capabilities of NCL, and the design and use of full screen facilities using the Panel Services component. Later chapters discuss the Netmaster Database (NDB) facilities.

Some material in this guide refers to topics covered in greater detail in the *Management Services User's Guide* and the *Management Services Implementation and Administration Guide*. You will be directed to these manuals where appropriate.

The *Network Control Language Reference* contains descriptions of all NCL and &NDB verbs, built-in functions, and system variables, their syntax and usage.

How to Use this Guide

This guide describes how NCL is structured and used. It describes important components of the language and how these are used within NCL statements to drive your SOLVE systems.

The components; verbs, built-in functions, and system variables are listed. This should be of particular use to readers familiar with Management Services.

The manipulation of Netmaster Databases (NDBs) is described in Chapter 16, *NDB Concepts.* Step-by-step instructions for creating an NDB are provided, as well as examples of code to demonstrate how to best work with NDBs.

A glossary of NCL and Management Services terms is included at the back of this guide.

What You Need to Know Before Using NCL

This guide assumes that Management Services has already been installed on your mainframe. It assumes that you are already familiar with basic IBM computer concepts and terminology. Specific NCL concepts and terminology are described where appropriate in the text, usually when first mentioned.

- Chapters 1 to 15 describe important elements of the NCL language and how these can be used to drive your various SOLVE systems.
- Chapters 16 to 21 describe how to create and use Netmaster Databases (NDB)s.
- The remainder of the manual, consisting of Appendixes A to I, is a reference section, which includes lists of verbs, built-in functions, and system variables.

Related Documentation

This guide and the *Network Control Language Reference* are the principal reference documents for users of NCL.

The *Network Control Language Reference* contains details of all NCL verbs, built-in functions, and system variables, their syntax and usage.

For information on installing, implementing, and using Management Services see:

- *Network*IT *and SOLVE Installation and Setup Instructions* (P01-240)
- *Management Services Implementation and Administration Guide* (P01-045)
- *Management Services User's Guide* (P01-020)
- *Management Services Release and Migration Guide* (latest edition)
- *Management Services Command Reference* (P01-012)

1

About Network Control Language (NCL)

This chapter discusses the following topics:

- What Is Network Control Language (NCL)?
- Creating NCL Procedures
- Invoking and Cancelling NCL Procedures
- Exiting OCS During Execution
- Controlling Runaway Loops
- Listing Procedure Names (OS/VS Only)
- Listing the Contents of NCL Procedures

What Is Network Control Language (NCL)?

Network Control Language (NCL) is a high-level interpretive language integrated into many SOLVE system components. It provides a fast, comprehensive and advanced development tool to implement your site-specific requirements. You can use NCL to rapidly tailor Management Services to suit the needs of your environment. NCL is based on free-form statement syntax that can process both system and user-supplied data. Data is maintained in variables that can be manipulated and changed as required.

Collections of NCL statements, that can include system commands, are termed *procedures* and are stored in Partitioned Datasets (OS/VS), CMS files (VM), or Source Statement libraries (VSE), called procedure libraries. These procedure libraries can be edited and updated while your system is operational. Each NCL procedure is a separate member within a procedure library.

There is one principal procedure library (or concatenation of libraries) used by Management Services. In addition to this system library, individual users under OS/VS can be allocated an individual procedure library for their own use, as part of the definition of their user ID.

NCL procedures can take many forms. They can be a simple collection of comment statements that provide an effective means of on-line documentation. They can be a collection of Management Services commands in exactly the same format as entered from a terminal. They can be extended to include logical decision-making capabilities, the display of full-screen panels, and the use of file processing capabilities.

An NCL procedure can *call* or *nest* another procedure to improve modularity.

In addition, certain NCL procedures are reserved for performing special functions such as interfacing to unsolicited messages from VTAM (PPOPROC), intercepting and reacting to other messages sent to the user terminal (MSGPROC), and processing messages destined for the activity logs (LOGPROC).

Creating NCL Procedures

Procedures can be created while the system is operational. New procedures can be added to the procedure library, or existing procedures modified. Any changes made to a procedure become effective the next time the procedure is loaded.

Procedures can be created using either system utilities or an online editor such as TSO/ISPF, CMS Edit, or ICCF.

To create a new procedure you must first select a (unique) name by which the procedure is to be identified and by which it will later be executed. The name you select should be meaningful and where possible identify the type of function that the procedure performs. It is recommended that you establish naming conventions for this library to ensure consistent use of names across operational areas.

Procedures are created as a series of individual records. Although the format of statements within each record is flexible, certain syntax restrictions do apply. These are discussed in Chapter 3, *NCL Statement Types and Syntax.*

Commands, comments, and statements can be entered in either upper or lower case. Most commands are converted to upper case before execution. Comments are left unchanged.

Checking NCL Syntax

When you have written an NCL procedure, you can use the NCLCHECK command to verify the syntax and structure without executing the procedure. NCLCHECK loads and checks the procedure just as if it were to be executed and notifies you of major syntax or structural errors in your code. Full syntax checking does not take place until execution.

Testing NCL Procedures

To ensure that system overhead is kept to a minimum, the system attempts to optimize NCL execution in many ways. One technique is that of *sharing* concurrent requests for a procedure of the same name, in such a way that only one copy of the procedure is loaded into storage. This technique also allows you to specify a *retention* queue on which a completed procedure is retained, on the assumption that it can be reused within a short period. This eliminates the need to perform disk I/O to bring the procedure into storage again.

For extremely high usage procedures, you can use a technique known as *preloading* to ensure that the procedure remains in storage at all times, regardless of its pattern of use. These options are selected by the SYSPARMS NCLPRSHR and PRELOAD operands. (For more information about the SYSPARMS command see the *Management Services Implementation and Administration Guide*.)

While such facilities are ideal in a production environment they can interfere with testing. For example, you may have completed and tested a procedure and made certain corrections and wish to run a second test. Although you have saved the corrected procedure in its library, you may notice the original errors are still occurring. This is most likely to be caused by NCL having retained the original procedure in storage and reused it when the second test was run. To overcome this apparent interference with your testing, use the NCLTEST command to signal to the system that you are in fact operating in a test mode and do not want your procedures retained or shared and that the latest copy is always to be loaded from the library. (For more information about the NCLTEST command, see the *Management Services Command Reference.*)

Debugging an NCL Procedure

The NCL Debug facility is a powerful tool to assist in the debugging of NCL procedures.

NCL Debug provides the ability to observe the execution of an NCL procedure from an external source—that is, another environment, another user region, window, or NCL environment. It eliminates the need for code changes in order to debug a procedure. It also provides comprehensive control over the NCL procedure as it is being executed, supporting statement stepping, alteration of variable contents and attributes, and so on. It allows you to specify criteria for debugging NCL before it begins execution.

The NCL Debug facility is made up of a set of commands that allows you to start and stop an NCL debug session, control the execution of NCL processes, and display and modify the contents of NCL variables. Procedure and subroutine nesting levels can be listed and the source code that is being executed can be displayed. NCL trace output can be received at another region, thus allowing you to view the trace output concurrently as the debugged process executes.

Invoking and Cancelling NCL Procedures

NCL procedures are invoked and cancelled by using MS commands.

Invoking NCL Procedures

Procedures are invoked either explicitly by using the EXEC and START commands, or implicitly by certain system functions such as EASINET and timer commands.

Optionally, variable parameters can be passed to the procedure when it is invoked. These variables are specified on the EXEC or START command following the name of the procedure being invoked. There is no limit to the number of variables that can be passed. Each word following the procedure name is passed in the next available variable, numbering from &1 for the first word, &2 for the second, and so on.

In VM systems the CMS minidisk that holds the Management Services procedures must be reaccessed after procedures have been altered, so that Management Services locates the new versions of the changed procedures.

Cancelling NCL Procedures

NCL procedures can be cancelled by using the FLUSH or END commands. Generally users can cancel only their own procedures, but the system can be configured so that authorized users can cancel any procedure. Full-screen procedures (that is, procedures that display full-screen panels) can normally be cancelled by using the standard END function keys (F3/4 or F15/16). If function key interception is being performed by the procedure, the responsibility of responding correctly to the use of function keys rests with the author of the procedure.

Exiting OCS During Execution

When an Operator Console Services (OCS) operator exits OCS, any NCL processes still executing in their OCS window are flushed without further execution and any display lines awaiting output are discarded. Any processes that are queued for execution by that user are also flushed. Messages are written to the log identifying any procedures that have been flushed.

Controlling Runaway Loops

Protection against uncontrolled looping within a procedure is provided through a loop control counter, maintained automatically by NCL. Runaway loop control can be activated by using &CONTROL LOOPCHK verb. The default is &CONTROL NOLOOPCHK, which means that runaway loop control is not in effect unless requested.

When a procedure commences execution, a count of 1000 is assigned to this counter. This is then decremented by one for each executed &DOEND statement associated with an &DOWHILE or &DOUNTIL, and for each executed &GOTO statement. If the count reaches zero, the procedure is regarded as being in an uncontrolled loop and is automatically terminated.

Certain events that involve operator interaction, indicating that the procedure is not looping, cause this counter to be automatically reset. An example of this occurs when displaying a panel using the &PANEL statement and then having to wait for operator input before processing can continue. In such a case the value for the loop control counter is reset to 1000. The &LOOPCTL verb is provided to enable the user to set a new loop control limit from within the procedure for those procedures that require more than the default value.

Note

&LOOPCTL is decremented for PPOPROC, MSGPROC, LOGPROC, AOMPROC, or CNMPROC. However, it is reset to its full value each time a message is read with &PPOREAD, &MSGREAD, &LOGREAD, &AOMREAD, &CNMREAD, or &INTREAD. The loop control therefore applies only for processing associated with one message.

Listing Procedure Names (OS/VS Only)

The SHOW EXEC command can be used to obtain an online list of the names of procedures in the procedure library. Optionally, a specified range can be listed:

```
SHOW EXEC,CA,CD
```

or the procedures in a particular procedure library can be listed:

```
SHOW EXEC,ID=PROCLIB2
```

For more information about the SHOW EXEC command see the *Management Services Command Reference.*

Listing the Contents of NCL Procedures

The LIST command displays the contents of the nominated procedure. No statements or commands within the procedure are executed. Nested procedures are not listed, and separate LIST commands are required if these need to be displayed.

2

NCL Concepts

Before attempting to write any NCL procedures it is important to understand how and where Management Services executes NCL procedures for you. This chapter introduces the major concepts of NCL execution and the terms used to describe NCL concepts throughout this manual.Other terms will also be used throughout this chapter and will be defined and explained as they appear.

These concepts are fundamental to understanding how NCL is integrated into the SOLVE system and how you can design NCL systems of your own that will run under Management Services.

This chapter discusses the following topics:

Where Does NCL Execute?

NCL executes only in association with a real or internally-simulated terminal. The most flexible method of invoking NCL processing, and the one that is easiest to understand, is to use the EXEC or START commands from an OCS window. These commands let you specify the name of the NCL program that you want to run.

Other methods of executing NCL are implicit rather than explicit, and are discussed later in this chapter.

What Is an NCL Procedure?

The basic unit of NCL code is called an NCL *procedure*. A procedure contains one or more NCL statements that are executed when the procedure is invoked.

Each NCL procedure has a 1- to 8-character name and resides in the Management Services procedure library. In OS/VS systems this is a Partitioned DataSet (PDS). In VM systems the library is held on a CMS minidisk and in VSE systems NCL procedures reside in source statement libraries under a specific sublibrary or member type.

Procedures are created or modified by an appropriate editor (for example, ISPF in MVS systems).

What Is an NCL Process?

The actual execution of an NCL procedure is termed an NCL *process*. The EXEC or START command specifies the name of an NCL procedure which is either retrieved from the NCL procedure library or is already resident in storage.

The following example shows the logical difference between a *procedure* and a *process*:

```
START      PROC1 NCPA
START      PROC1 NCPB
START      PROC1 NCPC
```

These commands, entered from an OCS window, invoke three separate NCL processes, each of which is executing procedure PROC1. There can be only one copy of PROC1 NCL code in storage, but three separate executions of it are taking place concurrently. In this example each process is given a different NCP name as a parameter.

It is nevertheless common for the term *procedure* to be used instead of *process*.

Nesting

The first procedure that is executed in a process can issue EXEC commands internally to call other procedures. This is called *nesting*. Regardless of how many other procedures are called, or the number of nesting levels involved, all execute as part of the same process. If any procedure in a process issues a START command, a new process is invoked, which is independent of the originating process.

Execution of the procedures in a process continues according to the logic of the procedures. When the first level procedure completes its processing, it ends and the process terminates.

The NCL Process Identifier

Each NCL process has a unique number associated with it when it starts execution. This is the NCL Process Identifier (NCLID), and it is a 6-digit number in the range 1 to 999999. Processes executing in a SOLVE system at the same time carry different identifiers.

The process identifier allows individual processes to be identified, even when processes are executing the same procedure.

The process identifier is used principally to specify which process is to be the target of a command such as GO, FLUSH, or INTQ, all of which can *talk* to NCL processes.

The NCL Processing Region

All NCL processing that occurs within a system is performed on behalf of Management Services users. These users can be people, who log on to the system with a user ID/password combination. Also, there are internal environments that *act* like real users, except that they do not have any real terminal associated with them. These internal environments all have *virtual* user IDs and in fact can be profiled in the same way as a real user ID by having a UAMS definition created.

The most common virtual user IDs are those associated with the *background environments* known as the *background logger* and *background monitor.* If you issue a SHOW USERS command from an OCS window, you will see these virtual user IDs listed along with any real users logged onto the system at the time. Other optional Management Services components generate their own background environments, so the list that you see on a SHOW USERS display varies according to the configuration of the system.

Regardless of whether a user is real or virtual, every user in a SOLVE system is capable of executing NCL processes, because every user has an NCL *processing region* associated with their user ID while they are logged on.

The NCL processing region provides all the internal services necessary to allow the user to have NCL processes executed on their behalf. While there is one NCL processing region for each user; within each user region there can be one or more NCL *processing environments*.

The NCL Processing Environment

Each real Management Services user is associated with a 3270-type terminal with a display screen that supports either one or two logical windows at any time.

Virtual users have no real terminals associated with them, but operate logically as if they own one real line mode window.

Associated with each window that a user operates is at least one *primary processing environment.* A processing environment provides the internal services and facilities that are required to execute NCL processes for the user from its associated *window.*

Real users, using real terminals, can have one or two windows and therefore can have one or two active NCL processing environments directly associated with these windows. Internal users have only one window (logically operating in line mode) and therefore have only one NCL processing environment.

An NCL process always operates within the NCL processing environment associated with the window from which it was invoked.

Executing NCL Processes Serially

You can use the EXEC command to invoke an NCL process which will execute *serially* with respect to other NCL processes that are invoked by the EXEC command from the same window (that is, in the same processing environment).

This means that if you enter:

```
EXEC        PROC1
EXEC        PROC2
```

from the OCS window command line, a process is invoked to execute procedure PROC1 immediately; the process invoked to execute PROC2 waits until the first process ends before starting to execute. The use of the EXEC command therefore *serializes* execution of NCL processes in each NCL environment.

Serial execution of processes is useful for handling sequences of functions or operations.

Remember: Each NCL processing environment is independent; each one can be executing its own stream of serialized processes. There is no limit to the number of processes that can be queued for serial execution within an NCL processing environment.

Executing NCL Processes Concurrently

You can use the START command to invoke an NCL process that will execute *concurrently* with other NCL processes that are invoked with the EXEC or START command from the same window (that is, in the same processing environment).

This means that if you enter:

```
START     PROC1
START     PROC2
```

from the OCS window command line, a process is invoked to execute the procedure PROC1 immediately, and the second process is also invoked to execute PROC2 immediately. You can use the START command to execute many independent NCL processes at the same time in the same NCL environment.

Concurrent execution of processes allows you to have *slave* processes running on your behalf doing different tasks. For example, a process could execute NCL procedures that monitor the status of particular resources at regular intervals and communicate with your OCS window only when something is found to be wrong.

The default maximum number of NCL processes that can be executing concurrently for any user is 128. You can change this by using the SYSPARMS NCLUMAX command.

The concurrent stream of NCL processes executes in parallel with the serial stream.

If you enter:

```
EXEC      PROC1
EXEC      PROC2
START     PROC3
START     PROC4
```

from the OCS window command line, PROC1 executes at once on the serial stream and PROC2 is queued waiting for PROC1 to finish. However, PROC3 and PROC4 start at once on the concurrent stream and therefore execute at the same time as PROC1.

The Dependent Processing Environment

The preceding sections of this chapter have outlined the concepts of the processing region (one per user) and the processing environment (at least one per window).

Processing environments can operate in a hierarchy, so that there can be many processing environments associated with the same window. This is achieved by using the &INTCMD NCL verb.

Every NCL process executing within an NCL processing environment can establish a *dependent processing environment* (using the &INTCMD verb) in which other NCL processes can be invoked to execute on behalf of the higher level.

In turn, an NCL process executing in a dependent processing environment can establish its own dependent environment, forming a hierarchy of dependency and allowing any NCL process to invoke chains of other dependent processes on its behalf.

The &INTCMD Verb

&INTCMD lets a procedure issue commands or execute other NCL processes and have the results returned to it rather than returning to the user's terminal window. This lets users write sophisticated procedures that check the results of their actions and correlate commands with their results.

When an &INTCMD statement is executed by a procedure, a new NCL processing environment is created that is subordinate to the process that owns it—that is, the process from which the &INTCMD statement is issued. This new processing environment is called a dependent processing environment, because it exists only until its originating process ends or issues an &INTCLEAR verb. The originating process can use &INTCMD to schedule commands or other NCL processes for execution within its dependent processing environment.

Any NCL process can issue &INTCMD and create its own dependent processing environment. As described before, if you issue the commands:

```
START     PROC1
START     PROC2
```

from your OCS window command line, processes PROC1 and PROC2 start executing in your window's NCL processing environment.

If each of those processes issues:

```
&INTCMD     START PROC3
```

then PROC1 and PROC2 each have a dependent processing environment in which a process PROC3 is executing. Similarly, the PROC3 processes can issue their own &INTCMD statements to create and use their own dependent environments.

Any process that ends, or that issues an &INTCLEAR statement, automatically causes the termination of all dependent environments, and therefore causes termination of whatever hierarchy of processes exists below it.

Explicit NCL Process Execution

The examples shown in the preceding sections have all assumed that the NCL processes involved have been executed as a result of a START or EXEC command. Using START or EXEC to invoke NCL processes is called *explicit execution*.

Implicit NCL Execution

Certain system functions execute NCL processes on behalf of users. The Primary Menu NCL procedure is an example of this. When a user logs on to Management Services, the NCL procedure nominated on the SYSPARMS MENUPROC command is executed.

In this case, NCL execution has been invoked *implicitly* as a result of a user logging on to Management Services, rather than as a result of a START or EXEC command.

Other examples of implicit NCL execution are:

- MAI menu procedure (MAI selected from the Primary Menu)
- EASINET procedure (invoked on behalf of each terminal when it connects)

System Level Procedures

Implicit NCL execution also occurs in relation to the various system level procedures that operate internally. The most common examples are:

LOGPROC

This process executes the NCL procedure nominated on the SYSPARMS LOGPROC= command. The process executes in the processing region of the LOGP user ID type, which is a virtual user. LOGPROC views, and can act on, all messages that flow to the log.

PPOPROC

This process executes the NCL procedure nominated on the SYSPARMS PPOPROC= command. The process executes in the processing region of the PPOP user ID type, which is a virtual user. PPOPROC views, and can act on, all unsolicited messages that are sent to Management Services by VTAM to report network events.

Other system level procedures can be started by optional components, for example, the Network Error Warning System (CNMPROC) and the Advanced Operation Management feature (AOMPROC).

The MSGPROC Procedure

A user operating in OCS mode can have a MSGPROC procedure associated with the window.

The MSGPROC procedure is invoked automatically when the user selects OCS mode from the Primary Menu, and is therefore *implicitly* executed. The MSGPROC procedure has access to all messages that are sent to the OCS window and can intercept, change or delete them as required. Message attributes such as color and highlight options can also be changed.

MSGPROC facilities are available to OCS windows running on real terminals, to background environments, system level procedures, and console environments. Processes that execute in dependent processing environments cannot have MSGPROCs associated with them.

Issuing Commands from an NCL Process

Each NCL process executes within the NCL processing region of the user ID that invoked it and is entitled to execute any command the user can execute from an OCS window command line. The process therefore has the same command authority as the user for which it is executing.

If a process executes a command, the rules outlined in the following sections dictate where the results of the command are sent.

In-line Command Execution

If an OCS operator enters:

```
START       PROC1
```

from the OCS window command line, the process PROC1 starts running.

If the process then executes the command:

```
SHOW USERS
```

the results of the command (the answer) flow to the OCS window. The process itself never has access to the messages generated as a result of the command. Consequently the procedure cannot make any decisions based on the results of the command.

The execution of a command by a process in this example is called *in-line command execution*.

Dependent Command Execution

If PROC1 is started in the same way as described in the previous section but executes the NCL statement:

```
&INTCMD SHOW USERS
```

the SHOW USERS command is executed in the dependent processing environment for the process. The messages generated by the command are not returned to the user's OCS window but queued in a stack called the *dependent response queue*. The process PROC1 can then issue the NCL statement:

```
&INTREAD ARGS
```

to read the results of the command from the dependent response queue one message at a time. The individual words of each message are tokenized and placed in variables of the form &1 &2... &*n*.

This technique lets a process issue a command and get the results back internally, so that the process can review the results and therefore make a decision based on the results of the command. This allows correlation of commands and results which in turn provides unlimited capability for complex logic to be built into processes to handle monitoring and automation of events.

The use of &INTCMD to execute commands privately in this way is called *dependent execution* of commands.

Review of Message Delivery Rules

The principal difference between in-line and dependent command execution is the message delivery that applies to the two different techniques.

A process that issues an *in-line command* never sees the results of the command—the results are always returned to the owner of the NCL processing environment in which the process is running.

A process that issues a *dependent command* sees the results of the command because they are queued to its dependent response queue and can be read using the &INTREAD statement.

NCL Processes and the Remote Operator Facility (ROF)

The concepts of NCL operation discussed so far have covered execution of processes within the NCL processing region of the user, in the system where the user is physically logged on.

ROF is a Management Services function that allows a user, who is logged on in one system, to route commands to other systems for execution. The results of these commands are then returned to the originating user.

ROF also allows a command to be routed to one SOLVE system for onward propagation to another SOLVE system where the command is to be actually executed.

Since ROF provides services at a command level, NCL processes (which can issue commands) are also entitled to use ROF services to route commands to a remote system and retrieve the results.

Message Flow on a ROF Session

A user who issues a SIGNON command to another system or who issues a ROUTE command to send a command to another system for execution establishes a *ROF session*. The user is logically logged on to the remote system and has user attributes and privileges as defined to the remote system.

If you enter:

```
ROUTE SOL2 SHOW USERS
```

from the OCS window command line, the SHOW USERS command is sent to the remote SOLVE system known as SOL2 and executed under your ROF logon. The results of the command are returned and displayed on your real OCS window.

If you enter:

```
START PROC1
```

to start the process PROC1, which in turn issues the in-line command:

```
ROUTE SOL2 SHOW USERS
```

the results are also returned to the real OCS window. In other words, the delivery of the results of in-line commands is the same across a ROF session as it is within the one system. If a process executes an in-line command either in its own system or by routing the command to another system, the results return to the owner of the NCL processing environment in which the process is executing; they do not return to the process itself.

Alternatively, if the process in the example issues the statement:

```
&INTCMD ROUTE SOL2 SHOW USERS
```

the results return to the PROC1 dependent response queue, and can be read back by PROC1 using the &INTREAD statement.

In summary, therefore, the delivery of command results is always the same, regardless of whether a process issues a command for execution in its own SOLVE system or a remote one.

The ability of NCL processes to issue commands across ROF sessions, and to correlate the results, allows the development of monitoring and control processes that can operate unseen and communicate with the operator only when a problem occurs.

Communication Between Processes

NCL processes execute in isolation from one another and although many processes can run concurrently in the same NCL environment or region, they usually have no knowledge of each other's existence.

However, it is often very useful to communicate directly with another NCL process either within the same NCL region or (depending on implementation options and user authority) across NCL regions.

The ability of processes to talk to each other provides the framework for developing co-operative processes that can coordinate their activities but remain independent. ROF services can also be used to provide communication between processes in different SOLVE systems.

The INTQUE Command

As described in the preceding sections, a process receives the results of commands that it issues using the &INTREAD verb. In concept there is a queue associated with the process, to which can be added messages that represent the results of commands. The &INTREAD statement allows the process to take messages off the queue one by one and process them as required.

The queue used for the messages generated by commands is called the *dependent response queue*.

The INTQUE command can also be used to place messages on a process's dependent response queue either by entering the command from an OCS window command entry line, or (more commonly) by issuing the INTQUE command from a different process.

The following example illustrates the concept. An OCS operator enters:

```
START PROC1
```

from the OCS window, which starts the process PROC1.

PROC1 then executes the following statements:

```
&WRITE DATA=&ZNCLID READY FOR WORK.
&INTREAD ARGS
```

At this point the process suspends execution pending the arrival of message(s) on its dependent response queue.

The operator, or another NCL process, then enters:

```
INTQUE ID=357 TYPE=RESP DATA=BEGIN
```

where 357 is the NCL process identifier of the process PROC1 and TYPE=RESP indicates that the text of the command is to be placed on the PROC1 dependent response queue. The &INTREAD of the process then completes, with the variable &1 containing the word BEGIN. The process can then go ahead with whatever other processing is required, having used the &INTREAD and INTQUE combination to provide direct communication between the process and the operator.

The Dependent Request Queue

While the INTQUE command can be used to place messages on the dependent response queue of a target process, it is sometimes confusing if messages arrive from an INTQUE command when the process is also expecting the results of a command it has issued. The INTQUE messages might arrive in the middle of the messages generated by the command.

To avoid this and to provide a method of isolating messages generated by commands issued by a process (which are predictable) from messages arriving from INTQUE commands issued externally (which are unpredictable), a second message stream is available. This is the *dependent request queue*.

If the example in the previous section is changed so that PROC1 executes:

```
&WRITE DATA=&ZNCLID READY FOR WORK.
&INTREAD ARGS TYPE=REQ
```

and the operator enters:

```
INTQUE ID=357 TYPE=REQ DATA=BEGIN
```

then the &INTREAD issued by PROC1 completes as before with &1 = BEGIN, but the communication flow takes place on the *request* flow rather than the *response* flow.

In a more complex scenario, PROC1 can use this differentiation in message flows to keep separate the results of its own commands (the response flow) and the arrival of messages from operators or other processes that arrive on the request flow.

Request and Response Disciplines

It is important to understand that the terms *request* and *response* are arbitrary and have no absolute meaning. They serve only to provide a logical separation between types of messages that can be used to simplify communications between processes.

A process that issues the statement:

```
&INTREAD TYPE=RESP ARGS
```

can never receive messages sent to it by a command:

```
INTQUE TYPE=REQ
```

If you wish to use INTQUE to send messages from one process to another, you must plan what you are going to send, make sure that the process you send a message to knows what to expect and is prepared to receive it, and that you coordinate the INTQUE and &INTREAD statements to make sure that they both expect the messages on the same request or response flow.

INTQUE Across ROF Sessions

Since INTQUE is a standard command, a process in one SOLVE system can use INTQUE across a ROF session to talk directly to a process on the remote system. To do this, the process (or OCS operator) must know the target process identifier in the remote system and ROUTE an INTQUE command to the remote system.

This delivers an INTQUE message to the dependent response or request queue of the target process, as if the target were executing in the same system.

If the target process is executing in the remote system as a result of a ROF command, that is, if it was started in the remote system by the command:

```
ROUTE SOL2 START PROC2
```

then PROC2 cannot use INTQUE to talk back to the local system. Instead, it uses &WRITE, which in a ROF environment causes all the command output generated by a remote process to flow back to the user ID environment in the local system.

The Scope of the NCL Processing Region

The concept of the NCL region is designed to limit the activities of individual users so that the NCL processes that they can run or communicate with are always within the user's own region. This prevents one user from affecting or tampering with NCL processes in use by other users or background environments.

If the SYSPARMS NCLXUSER operand is set, the system can be configured to allow *cross-NCL region* communications by users with the appropriate authority.

In conjunction with the NCLXUSER operand, which determines whether cross-region communication is allowed, the INTQUE command has a second authority level that is checked when a user attempts to route a message to a process outside their own NCL Region. The user must have an authority level equal to or greater than the *opauth* value assigned to the INTQUE command. The *opauth* value defaults to authority level 2, but could be different in your installation.

Finding Out Which NCL Processes Are Executing

Use the SHOW NCL command to display the status of active NCL processes. SHOW NCL provides displays of NCL activity by *environment* or by *region*:

- Environment shows you details of NCL processes that are executing in the NCL environment of the window from which the command is entered
- Region shows you all NCL activity associated with your user ID

SHOW NCL can also be used to display the status of NCL processes executing on behalf of other user IDs if you have sufficient command authority. See your *Management Services User's Guide* for further details.

3

NCL Statement Types and Syntax

An NCL procedure comprises a group of NCL *statements* that describe the logic and functions to be executed when the procedure is invoked. There are different types of NCL statements, where the type is determined by the function that the statement performs within the procedure.

This chapter describes the following topics:

- Format of NCL Statements
- NCL Conventions and Syntax
- Comments in NCL Procedures
- Label Statements
- Verb Statements
- Built-in Function Statements
- Assignment Statements
- Command Statements

Format of NCL Statements

NCL statements are coded as free-format, 80-character records where the first 72 characters contain the statement syntax and the last 8 characters can contain optional statement sequence numbers that, if present, are used in error messages generated by NCL.

An NCL procedure can contain statements that are completely blank. These are useful for visual layout when reading a procedure and are ignored when the procedure is executed.

Statement Continuations

Statements could require more than the 72 characters available in a single record. To accommodate this, NCL supports the continuation of a statement across multiple consecutive lines. Use of continuations can assist in improving the layout of a procedure and simplify future modification.

The number of continuations is limited to a total statement length of 2048 characters, before variable substitution.

A continuation is indicated when the last non-blank character on a statement (not including the optional sequence number in columns 73 to 80) is a plus (+) sign.

When a continuation is detected, the plus sign is removed and the text of the next statement is concatenated to the statement that contained the plus sign, after stripping leading blanks. This concatenation continues until a statement is found that does not have a plus sign as the last non-blank character. For example:

```
&FILE GET ID=MYFILE OPT=KEQ VARS=(A,B,C,+-* DATE, TIME
                                       -* and NAME
                D,E,F,G,+              -* ADDRESS
                                       -* DETAILS
                H,I,J)                 -* EQUIPMENT
                                      -* DETAILS
```

would result in the statement:

```
&FILE GET ID=MYFILE OPT=KEQ VARS=(A,B,C,D,E,F,G,H,I,J)
```

Note

Continuation is deactivated by the &CONTROL NOCONT statement. This might be necessary in procedures where a plus sign is used, for example, in setting a function key where the plus sign also indicates an implied blank. In such cases, these statements should be preceded by an &CONTROL statement that deactivates continuation processing and followed by another that reactivates it. For example:

```
&CONTROL NOCONT
PF5 PREF,MSG USER1+
&CONTROL CONT
```

The CONT and NOCONT operands of the &CONTROL verb cannot be coded as variables.

Variable Substitution

Statements can contain *variables* that have contents that are resolved through a substitution process at the time the statement is executed. The use of variables within the statement syntax effectively changes the text of the statement by substituting the current value of the variables imbedded in the statement text each time the statement is executed.

During the substitution process, a restriction of a maximum word size of 256 characters exists. Additionally, the total size of a statement after substitution has been completed is 12288 characters.

For example, the length of the *coded* statement

```
&IF &A = &B &THEN &GOTO .END
```

is 28 characters, including blanks between the words. If it were used in the following manner:

```
&A = ABCDEFG
&B = ABCDEFG
&IF &A = &B &THEN &GOTO .END
```

then, after the substitution process performed at the time the statement is executed, the statement would read:

```
&IF ABCDEFG = ABCDEFG &THEN &GOTO .END
```

and be 38 characters long.

Variable substitution within statements allows you to code a standard logic routine that processes different information.

NCL Conventions and Syntax

The logic capabilities of procedures include the ability to define user and global variables, test the truth of conditions, and to make conditional branches on the basis of these tests. To achieve this, certain syntax rules must be obeyed:

- Procedure statements are stored in 80-character records. Columns 1 to 72 can be used to contain commands, statements, or comments. Columns 73 to 80 are reserved for optional sequence numbers. If present, these sequence numbers are used in error messages to pinpoint a statement in error.
- Comments can be included within procedures to assist in documentation and you are encouraged to use them freely.

  The syntax for comments is discussed below. Comments can occur as separate statements or on other types of statement.
- Continuation of a statement onto the next record is supported if the last non-blank character on the line (excluding comments) is a plus sign (+). The total length of a statement including all continuations cannot exceed 2048 characters. &CONTROL CONT must be in effect.
- Syntax is free-format and can start in any column to improve presentation.
- A procedure can invoke other procedures. This process is called nesting. Nesting to 64 levels is supported. Variables can be passed to a procedure either when the procedure is invoked by the operator or when a nested procedure is invoked.
- No individual word or field within a statement can exceed 256 characters.
- The value of a variable cannot exceed 256 characters in length.
- The total length of an expanded statement after variable substitution must not be longer than 12288 characters.
- Statements can contain any number of leading or trailing blanks. A completely blank statement is ignored. No specific termination statement is required to signal the end of a procedure. When the end of the procedure is reached (end-of-file), the procedure terminates, unless terminated through other processing options (for example, &END) or errors.

Comments in NCL Procedures

NCL allows the inclusion of comments within procedures to assist in understanding the code. In addition to providing internal documentation of the processing being performed by the procedure, comments can also provide a simple means of directing messages to the operator when operating in OCS mode.

Comments fall into four categories:

- Comments on NCL statements
- Displayable stand-alone comment lines
- Highlighted key words in comment lines
- Non-displayable stand-alone comment lines

Comments on NCL Statements

Individual statements can include comments after the statement text, to describe the function being performed by that statement. In such cases the comments must commence with the two suppression characters, dash asterisk (`-*`). The text of the comment following the `-*` is free-form. For example:

```
&PANEL  TESTPANEL  -* DISPLAY OUR TEST PANEL.
```

Where a statement spans multiple lines and therefore contains continuations, comments can be included on each line if the comment text follows the continuation indicator.

```
&FILE GET ID=MYFILE OPT=SEQ +-* SEARCH IS TO BE
                            -* SEQUENTIAL
          VARS=(A,B,C)      -* RECEIVE INTO THESE
                            -* VARIABLES.
```

Displayable Stand-alone Comment Lines

Comment lines can be included in procedures by inserting an asterisk (*) as the first non-blank character in the statement, unless the statement contains a label, in which case the asterisk must be the first non-blank character following the end of the label. Comment lines are displayed at an OCS window exactly as entered (both upper and lower case) after variable substitution has been performed. The leading asterisk is not displayed. This facility provides a simple means of building online operator instructions or Help facilities. For example:

```
*BRING UP WESTERN SECTOR AT 13.00.
```

Highlighted Key Words in Comment Lines

Key words within a comment line can be highlighted. This is achieved by coding a plus sign (+) in place of the asterisk as the first non-blank character in the statement. Such lines are scanned for the occurrence of the highlight indicator, which is an at (@) sign . Words bounded by this character are displayed in high intensity and a blank substituted in place of the @ character. Multiple occurrences can exist within a line.

For example:

```
+BRING UP@WESTERN SECTOR@AT@13.00@.
```

will display as:

BRING UP **WESTERN SECTOR** AT **13.00.**

If only one @ is contained in a line, the remainder of that line is displayed in high intensity. The effect does not flow on to the next procedure statement.

Note

The &WRITE verb offers further facilities for displaying information and for using color and highlighting to improve information presentation.

Non-displayable Stand-alone Comment Lines

If the minus (-) suppression character (see next section) is used in conjunction with the asterisk (*) it indicates that the comment line display is to be suppressed.

In such a case the comment is not displayed, even if executing from an OCS window, and acts solely as a source of documentation within the procedure. For example:

```
-*
-* THIS PROCEDURE ACTIVATES THE NETWORK
-*
```

The Suppression Character

The minus character (-) is reserved as the suppression character. If used, the suppression character (-) must be the first non-blank character in the line. The suppression character can be used with both command and comment lines, but not NCL statement lines. When used in a command line, the command is executed in the normal way. However, the command is neither displayed (echoed) on your screen before execution, as would normally be the case, nor written to the system's activity log.

Additionally, the suppression character can be used during entry of commands from an OCS window. For example:

```
-SHOW USERS
```

Note

Any results generated by the execution of the command are displayed at your terminal and are written to the activity log.

An alternative to using the suppression character is the &CONTROL NOCMD statement. This statement prevents the echoing of all commands within the procedure until the end of the procedure or until and &CONTROL CMD statement. When using NOCMD, the results generated by commands are written to the activity log.

Use of &CONTROL NOCMD has no impact on comment lines and does not cause them to be suppressed.

Label Statements

NCL supports the definition of labels that can be branched to during processing using an &GOTO or &GOSUB statement. The following rules apply when defining labels:

- Labels commence with a period (.) and must be followed by 1 to 12 characters and not contain any ampersands (&).
- A label need not commence in column 1. However, it must be the first item on the statement.
- A label can be placed on a statement containing no other data.

Valid labels:

```
.LABEL1          -* Normal label
.10              -* Numeric label
.ACT-NODE        -* Label containing special characters
.CMD1 statement -* NCL statement following label
```

Invalid labels:

```
BAD              -* Does not commence with a period.
.LABELTOOOLONG   -* Label must be 1 to 12 characters.
.LABEL&1         -* Cannot contain ampersands.
XYZ .LABEL       -* Must be first item on line.
```

Label Variables

A variable can be specified as the target of an &GOTO or &GOSUB statement. This variable, after variable substitution, is then taken as a label to which a branch is required. For example:

```
&GOTO .&MSGID
```

can read, after variable substitution:

```
&GOTO .IST350I
```

and a search for the label .IST350I is made.

Usually, most information input to a procedure contains a unique identifier such as a message number. The ability to branch to a label variable makes it possible to use this unique message number and branch directly to the part of the procedure concerned with processing the message, instead of having to use a number of &IF statements to determine the processing required.

Undefined Labels

Normally, an attempt to branch to an undefined label results in an error and the procedure is terminated. However, the &CONTROL NOLABEL operand allows the user to specify that an attempt to branch to an undefined label will not result in an error, but will simply drop through and execute the statement following the &GOTO. Thus, a trap for unexpected messages can be set up and all of the efficiencies associated with direct branching still achieved. For example:

```
&CONTROL NOLABEL
.NEXT
     &PPOREAD VARS=(A,B,C,D,E)
     &GOTO .&A
     &WRITE ALARM=YES DATA=UNEXPECTED INPUT &A &B &C &D &E
     &GOTO .NEXT
.IST305I
.IST314I
     .
     .
     .
```

Duplicate Labels

A label in an NCL procedure should not appear more than once; NCL normally checks to ensure a label is not duplicated. If the &CONTROL NODUPCHK operand is used, duplicate label checking is not performed. This option can result in improved efficiency, but is normally only used after the procedure has been thoroughly debugged.

While label variables offer great benefits in both performance and simplicity, one point needs to be considered. By its very nature, a label must be unique within a given procedure. An attempt to &GOTO a duplicate label will result in an error (unless the &CONTROL NODUPCHK option is in effect). If the processing of a label variable yields a duplicate label, perhaps conflicting with one used earlier in the procedure, you can use the following—if you are using a variable in an &GOTO that is itself not unique, you can append one or more characters to the label in the &GOTO to ensure the generated label is in fact unique. In the following example, the operator can enter YES or NO to either of the &PAUSE statements. In the second, the letter X is prefixed to the operator's input to differentiate it from input entered in reply to the first &PAUSE statement.

```
&WRITE DATA=ENTER 'Yes' OR 'No'

.PAUSE &PAUSE ARGS
     &GOTO .&1
     &WRITE DATA=INVALID RESPONSE, RE-ENTER.
     &GOTO .PAUSE
     .NO &END
     .YES
     .
     .
     .
     &WRITE DATA=ENTER 'Yes' OR 'No'

.PAUSE2 &PAUSE ARGS
&GOTO .X&1

&WRITE DATA=INVALID RESPONSE, RE-ENTER.
&GOTO .PAUSE2          -* Note addition of X to both
.XNO &END              -* &GOTO and target labels to
.XYES                  -* make labels unique.
```

Minimizing Labels

By using the verbs &DO, &DOWHILE and &DOUNTIL in your procedure logic you reduce the need for &GOTO and &GOSUB statements. This is recommended, both to reduce the number of labels that you need in a procedure and to improve the structure of your procedures.

Verb Statements

NCL *verbs* cause actions to occur. There are different types of verbs, some that dictate the flow of processing and logic, others that fetch information for the procedure to process and others that cause data to flow to external targets.

Examples of verbs are:

&DOWHILE	causes processing to continue in a loop while a certain condition exists.
&WRITE	allows a procedure to write a message to the user's terminal.
&FILE	allows a procedure to read, write, or delete records on a file.

NCL statements that contain a verb have the general form:

```
verb [ opt=opt  &variable ...  &variable ]
```

where *verb* is the keyword that is the name of the verb to be executed, *opt* is a keyword operand identifying a subfunction of the verb, and &*variable* represents parameters required as input to the verb's actions, or the names of variables that receive the results of the verb's execution.

NCL verbs and built-in functions are listed in Appendix A, *NCL Verb and Built-in Function List*. They are fully described in the *Network Control Language Reference*.

Built-in Function Statements

NCL *built-in functions* perform commonly-required functions that are either impossible to achieve using NCL or require complex NCL coding.

Examples of built-in functions are:

&DEC	converts a hexadecimal number to its decimal equivalent.
&SUBSTR	allows you to extract part of one variable and place it in a second variable.
&CONCAT	allows you to concatenate the values of two or more source variables and place the result in a target variable.

NCL statements that contain a built-in function have the general form:

&*target* = *function parameter*

where &*target* is the name of a variable that is to receive the result of the built-in function, *function* is the keyword that is the name of the built-in function to be executed and *parameter* represents parameters required as input to the built-in function.

NCL verbs and built-in functions are listed in Appendix A, *NCL Verb and Built-in Function List*. They are fully described in the *Network Control Language Reference*.

Assignment Statements

Assignment statements let you set a new value for a nominated variable, and have the general form:

&*variable* = *expression*

where &*variable* is the name of the target variable, whose value is to be changed to the result of the execution of *expression*. The target variable is always on the left-hand side of the equals sign (=) and the value to be assigned to the target is always on the right-hand side of the = sign.

Examples are:

```
&A = 10                        -* &A is assigned the value 10
&XYZ = ABCDEF                  -* &XYZ receives the value ABCDEF
&A = (&C + 1) - (&B * 5)       -* Add 1 to &C then subtract
                               -* five times the value of
                               -* &B and put the result in &A.
```

Assignment statements are used to manipulate the values of the variables that your procedure uses.

Assignment also occurs as the result of executing built-in functions and some verbs. Assignment and the rules associated with it are described in Chapter 4, *Variables, Substitution, and Assignment*.

Arithmetic

Arithmetic is performed by a series of explicit assignment statements, in which the target variable receives the result of the calculation coded to the right of the = sign. Compound arithmetic expressions are supported by NCL, and both integer and floating-point arithmetic can be used. The rules and support for arithmetic functions are described in Chapter 5, *Arithmetic in NCL*.

Command Statements

You can execute any MS command within a procedure by coding the command statement exactly as you would enter the command from an OCS window.

You can also code the command statement to include variables in the command text, in which case substitution is performed before the command is executed.

4

Variables, Substitution, and Assignment

This chapter discusses the following topics:

- What Is a Variable?
- Variable Types
- DBCS Device Support
- Variable Substitution
- Complex Variable Substitution
- Aligning Substitution Data
- Lower Case Data
- Debugging Procedures
- Setting Variables to a Particular Value
- NCL Table Manipulation
- The Vartable Facility

What Is a Variable?

The term *variable* describes a word that may represent different values within a statement. A variable commences with an ampersand (&) and is followed by 1 to 12 characters that form the name of the variable.

A variable name can contain the characters $, #, @, A to Z, and 0 to 9. The name is delimited by the first blank or non-valid character. Although the names of variables can be coded to contain both upper and lower case characters (A to Z) they are treated internally as being entirely upper case.

However, when operating in a system where support for double byte character stream (DBCS) terminals is active, no translation to upper case is performed.

If a variable name *starts* with a digit, the *whole* variable name must be numeric.

Variable Types

There are four types of variables:

- System variables
- User variables
- Global variables
- Parameters

System Variables

A number of variables are maintained by the system to provide access to commonly required facilities. An example of a system variable is the time, which can be obtained by referencing the &TIME system variable.

User Variables

Any number of user variables (within storage limitations) can be defined to contain information required during the processing phase of a procedure. Explicit definition of user variables is not required. The first time a user variable is referenced it is automatically defined. User variables are automatically deleted when a procedure terminates.

User variables can be passed across nested procedure levels if the &CONTROL SHRVARS operand is in force when the nested procedure is invoked, otherwise they are unique within the current nesting level. If a user variable is required in a nested level and &CONTROL SHRVARS is not in effect, its value must be passed explicitly as a parameter on the EXEC command when the nested level is invoked. An &RETURN statement can then be used to return specified variables to a higher nesting level, thus facilitating the development of modular functions.

Global Variables

Global variables operate in a similar manner to user variables. The difference is that global variables, as the name implies, are global to the entire system and once created can be referenced or modified by any procedure. Thus, global variables can be used to perform cross-talk between procedures.

Global variables commence with a 1- to 4-character prefix that you can set for your site using the SYSPARMS NCLGLBL command. Unless otherwise defined this prefix defaults to GLBL. The remaining characters must conform to standard variable naming conventions.

The &000 system variable is set to the current value of the global variable prefix. Thus, if the SYSPARMS NCLGLBL=#@ command was used to set the global variable prefix to #@ then &000 returns that value. This facility is provided to allow the procedure designer to develop procedures that execute correctly regardless of the global variable prefix. For example, &&000SYS1 is resolved to &GLBLSYS1 using the default value for the global variable prefix, or &#@SYS1 if the prefix was set as mentioned above.

A global variable can be created explicitly by an assignment statement, for example:

```
&&000SHIFTLDR = &STR SHIFT LEADER IS BILL SMITH
```

or implicitly by specifying a global variable name as the target of a verb that creates or modifies variables as part of its function.

Once created, a global variable remains in existence until it is explicitly deleted by an assignment statement. For example:

```
&&000SHIFTLDR =
```

Note

The uncontrolled creation of large numbers of global variables can consume a large amount of storage. You should ensure that global variables are deleted as soon as they are no longer needed. Care should be taken when designing procedures to ensure that they do not end without deleting any redundant global variables created.

It is recommended that you observe a standard naming convention for global variables to avoid the inadvertent destruction of variables where, for example, one procedure creates a global variable with an identical name as that created by another.

The SHOW NCLGLBL command provides a display of the global variables that exist in the system.

Parameters

Parameters are those variables passed to the procedure when it is invoked by the operator, by another procedure or by Management Services itself.

Parameters entered when the procedure is invoked are positional and must be entered in the order expected by the procedure. Entered parameters are separated by one or more blanks. Commas are not accepted as delimiters between parameters. The procedure must verify that the entered parameters are correct. Each parameter is allocated to a variable in the form &*n*, where *n* is a number, starting at 1, identifying the position of the parameter.

If the operator enters:

```
EXEC TEST1 PU1 LU2 NCP3
```

the following variables are created:

```
&1  will be set to  PU1
&2  will be set to LU2
&3  will be set to  NCP3
```

In addition, when a procedure is invoked, the user variable &ALLPARMS will be set to the entire string provided, excluding the EXEC or START command and the procedure name. In the above example &ALLPARMS is set to:

```
PU1 LU2 NCP3
```

The system also supplies a count of the number of variables created in the system variable &PARMCNT. In the above example &PARMCNT is set to 3.

Note

&CONTROL SHRVARS can be utilized to pass some or all variables to a lower nesting level rather than specifying individual parameters.

DBCS Device Support

Management Services supports terminals that use DBCS representation of symbols for languages such as Japanese (Kanji script).

When DBCS support is active, the system no longer automatically translates lower case characters to upper case for comparative purposes and so on. When DBCS is active all characters in the extended character set are regarded as unique, and certain system functions no longer apply, or are treated as a *no-operation*. Impacted facilities are documented within the relevant sections of this manual.

Variable Substitution

Before a procedure statement is analyzed or a command is executed, the record is scanned for the presence of variables, indicated by an ampersand (&). This process is called *variable substitution*. Variable substitution is performed from right to left of the statement. When a potential variable is detected by the presence of an ampersand, the variable is isolated by scanning to the right for a delimiting blank, comma or other character not valid in a variable name. Note that a single character ampersand is not treated as a variable and remains intact as a single ampersand.

Take the string &A.1 as an example. &A is processed as the variable and the resulting substitution inserted in its place (for example, 123.1).

Variables whose names are entirely numeric (for example, &99) have unique properties in that they are delimited by the first non-numeric character. For example:

```
&91 = 123
&A  = &91ABC
```

The variable &A in this example would yield the value 123ABC. This technique can provide an alternative to the use of &CONCAT with alphanumeric variables.

In the substitution process no spaces are added or deleted. The variable is removed and the text to the right of the variable is relocated to accommodate the data being substituted in place of the variable. A variable can be referenced as often as required. Left-hand and right-hand alignment of substituted data can be achieved using the &CONTROL ALIGNL and ALIGNR options to enhance tabular output or when displaying data in full-screen panels the #FLD statement can be used to assign specific alignment attributes to a field.

Undefined variables, or variables that have a null value, are eliminated from the line. You should therefore take care to ensure that variables are correctly coded and will not be eliminated during the substitution process. For example, the statement:

```
&WRITE DATA=The time is &TIMW and date is &DATE1
```

incorrectly contains the variable &TIMW where &TIME was intended. As &TIMW does not have an assigned value, it is eliminated from the statement; the final result written to the user's terminal is:

```
The time is  and date is 91.001
```

In certain cases, such as with an &IF statement, this elimination of variables can pose problems and result in syntax errors. Consider the following statement:

```
&IF &1 EQ YES &THEN &GOTO .OK
```

If &1 is a null value, perhaps because an operator did not enter its value after an &PAUSE, the statement after substitution appears as:

```
&IF EQ YES &THEN &GOTO .OK
```

and results in a syntax error and the procedure terminates.

A technique that can be used to avoid this is to append a constant character to the variable and, of course, to the value to which it is to be compared. For example:

```
&IF .&1 EQ .YES &THEN &GOTO .OK
```

In this case, if the variable &1 has a null value, the following statement results:

```
&IF .  EQ .YES &THEN &GOTO .OK
```

which is syntactically correct and performs as desired.

Note

When comparing numeric values, a zero (0) can be used as the constant character.

See the &IF verb description in the *Network Control Language Reference* for further discussion on using this technique.

Complex Variable Substitution

Complex or multi-level variables are supported. If after one substitution, the value generated is still a variable (commences with an &) substitution is again performed. Take the following example:

```
&A = 1
&1 = MESSAGE
&WRITE DATA=TEST &&A
```

Because the current value of &A (1) is substituted into the statement, the first pass of the &WRITE statement yields:

```
&WRITE DATA=TEST &1
```

The second pass then resolves &1, and yields:

```
&WRITE DATA=TEST MESSAGE
```

Complex variables are important when performing matrix processing where an unknown number of variables must be dynamically generated to accommodate data. For example:

```
&FILE OPEN ID=MYFILE
&FILE SET ID=MYFILE KEY=MYKEY
&LIMIT = 100
&CNT = 1
&DOWHILE &CNT LT &LIMIT
          &FILE GET ID=MYFILE OPT=KEQ VARS=MYFIELD1
          &SAVE&CNT = &MYFIELD1
          &CNT = &CNT + 1
&DOEND
```

Aligning Substitution Data

The substitution process does not, by default, preserve any difference in the length between a variable name and the data being substituted in place of the variable. This can make tabular output of rows of numbers difficult or impossible to align when not using full-screen facilities.

The &CONTROL statement provides the ALIGNL and ALIGNR options to specify special alignment requirements. ALIGNL requests left-hand alignment and ALIGNR right-hand alignment. Both of these options can additionally identify a fill character (by default a blank) that is to be used, for example, ALIGNR$.

When using these alignment options, the length of the name of the variable itself is used to determine both the point of alignment and the number of fill characters required. Therefore variables with names of the same length must be used to ensure the correct alignment is achieved.

In the following example the three variables &COUNT01, &COUNT02 and &COUNT03 are used to display a table of numbers. They are to be right-aligned and padded with leading dashes (-).

```
&CONTROL ALIGNR-
&COUNT01 = 1
&COUNT02 = 1098
&COUNT03 = 66
&WRITE DATA=COUNT 1 = &COUNT01
&WRITE DATA=COUNT 2 = &COUNT02
&WRITE DATA=COUNT 3 = &COUNT03
```

The result is:

```
COUNT 1 = -------1
COUNT 2 = ----1098
COUNT 3 = ------66
```

Note

These techniques are designed to be used by NCL procedures running in the OCS environment which do not utilize full-screen panels. See *Chapter 6, Designing Interactive Panels (Panel Services),* for specific techniques to use with full-screen panels.

Lower Case Data

By default, data assigned to a variable is converted to upper case. However, use of the &CONTROL NOUCASE statement allows lower case data, such as input from a full-screen panel, to be maintained. If you want comparison operations to occur without translation of the operands being compared, specify &CONTROL NOIFCASE, but note that the NOUCASE option must also be in effect.

Note

For data to be entered into a panel in lower case the field definition for the panel must also specify CAPS=NO.

When operating in a system with the SYSPARMS DBCS=YES option specified, no distinction is made between upper and lower case and the rules regarding automatic conversion to upper case do not apply.

Debugging Procedures

During the development of procedures coding errors are bound to occur. The &CONTROL TRACE and &CONTROL TRACELOG statements allow you to request statements to be written to your terminal or the system log after variable substitution and before execution. The &CONTROL statement can occur as frequently as required and can therefore be placed strategically within the procedure.

The TRACELAB and TRACEALL options are available to provide label flow tracing and to include additional detail on the trace records that are logged.

The &CONTROL NOTRACE statement returns to normal processing. You will have imposed a limit to the maximum number of trace messages that can be generated by a procedure. By default this limit is 100 messages and has been imposed to stop excessive tracing consuming system resources.

The NCLTRACE command can be used from an OCS window to dynamically start or stop NCL procedure tracing while an NCL procedure is executing. This lets you debug executing processes without having to stop them, edit the NCL to include &CONTROL statements, and then restart them.

The NCLTRACE command is described in the *Management Services Command Reference* and the &CONTROL verb is described in the *Network Control Language Reference*.

When testing procedures that use full-screen processing, the temporary inclusion of variables in a vacant area in a panel can assist in tracking and development. These can be removed when the procedure has been completed.

The &WRITE statement can also be used to great advantage to display the contents of variables at specific points in processing.

Note

When using &WRITE from procedures operating in full-screen mode, the messages are queued and displayed in one or more full-screen panels when the procedure terminates. Having completed the display of any queued messages, the procedure ends.

See also the NCL Debug facility described in Chapter 15, *The NCL Debug Facility*.

Setting Variables to a Particular Value

Assignment is the term used for the process of setting variables to a particular value. Assignment is either explicit, in which case you use the assignment statement to change the value of a variable, or implicit, in which case the action of a verb causes a change in value of a target variable (or variables).

Explicit Assignment: The Assignment Statement

The basic format of the assignment statement is:

&*variable* = *expression*

&*variable*

Is the name of the *user variable* or *global variable* whose value is to be changed. The variable might not actually exist, in which case the assignment statement creates it then sets its value.

The name of the variable must be valid according to the rules for variable names described earlier in this chapter.

If a *system variable* is specified as the target of an assignment statement a syntax error will occur.

Complex variables are supported and are described below.

=

Is written as shown, indicating that an assignment is required.

expression

Represents the value that is to be assigned to the target variable. If no *expression* is supplied the target variable is assigned a null value, and the variable is automatically deleted. An *expression* is one of the following constructs:

- A variable or constant whose value is to be assigned to the target variable.
- A built-in function such as &CONCAT, &SUBSTR, and so on. When a built-in function executes it produces a result that is used as the value to be assigned to the target variable.
- An *arithmetic expression.* The arithmetic calculation is performed to yield a result, which is then used as the value to be assigned to the target variable. The format and rules for arithmetic expressions are described in Chapter 5, *Arithmetic in NCL*.

Using Complex Variables in Assignment Statements

Use of complex variables within an assignment is supported. A complex variable requires multiple substitutions to determine the final variable required. For example:

```
&&1
&A&1
```

An example of the use of complex variables is the accumulation of counts for different values. A counter is initialized for each possible value. A single statement can then be used to increment the associated counter.

For example, assume that a variable (&1) can have the values A, B, or C and you want to count the occurrences of each of these. This could be achieved in the following way:

```
&CNTA = 0
&CNTB = 0
&CNTC = 0
   .
   .
   .
&DOWHILE &CNT&1 NE &LIMIT
   .
   .
   processing to receive input and set &1 to A, B or C
   .
   .
   &CNT&1 = &CNT&1 + 1
&DOEND
```

The assignment statement target (&CNT&1) will be resolved to &CNTA, &CNTB or &CNTC prior to execution.

Implicit Assignment: Using &ASSIGN

The &ASSIGN statement, which is an NCL *verb*, is used to assign values to multiple target variables in one operation. It is common for procedures to operate with groups of related variables and to have the need to manipulate the values of all the variables in a group.

Using explicit assignment statements to change the values of all the variables in a group is less efficient than using the &ASSIGN statement.

&ASSIGN lets you define named or generic groups of source variables, and transfer source variable contents to corresponding target variables, also named specifically or generically. Similarly, groups of target variables can be assigned null values and deleted in a single operation.

The &ASSIGN verb and its use are described in the *Network Control Language Reference.*

Implicit Assignment: Using Other NCL Verbs

While &ASSIGN is a verb designed specifically to allow multiple assignment operations, many other NCL verbs also cause assignment of a new value to one or more target variables as the result of their execution. Examples of these NCL verbs are &PARSE and &SETVARS. A complete description of these verbs and how to use them can be found in the *Network Control Language Reference.*

Verbs whose action results in retrieving information from an external source (for example, &FILE, &PAUSE) place the data they receive into specified target variables. An implicit assignment of values takes place as a result of the execution of this type of verb.

Upper and Lower Case Variables

By default, when character data is assigned to a target variable the data is converted to upper case.

If you want to manipulate lower case character data you must use the &CONTROL NOUCASE processing option within your procedure. The NOUCASE option stops the automatic translation of character data to upper case on assignment statements.

Remember that when your procedure starts executing the default &CONTROL option is UCASE, which will cause upper case translation.

DBCS Support and Lower Case Data

If your system is operating with DBCS support active (SYSPARMS DBCS=YES) then the system regards all data as a single case and no upper or lower case translation is performed. In this execution mode the &CONTROL UCASE option is ignored.

Variables and Storage Usage

Variables hold the data your procedures work with. This data occupies virtual storage in a Management Services region or partition. Therefore, the more variables you create the more storage your procedures will use.

While Management Services attempts to optimize storage usage, you should design your procedures to control the use of variables and avoid creating large numbers of variables that are seldom referenced.

If the temporary use of an extremely large number of variables is required, then assign a null value to these variables when they are no longer required to delete them and reduce the storage required by the procedure.

The &ASSIGN statement can be used to nullify large groups of variables in a single operation.

In MVS/XA and ESA systems all storage used for NCL variables resides above the 16 Mb line.

NCL Table Manipulation

The discussion of variables and substitution has so far concentrated on using simple and complex variables as single data entities. You can also construct tables of data using multiple individual variables using either of two common techniques:

- You can use complex variables, for example, &DATA&INDEX where the variable &INDEX contains a number (for example, 1,2,3), so generating variable names of &DATA1, &DATA2, &DATA3, and so on.

  This technique is fairly efficient for small tables, but the overheads of creating and managing large numbers of variables become significant when dealing with large tables. However, it is only suitable for numeric table indexes, or short alphanumeric indexes that do not contain any characters that are invalid in an NCL variable name.
- The second technique uses the complex variable approach described above, but each entry in the table is represented by 2 (or more) variables (for example, &KEY&INDEX, and &DATA&INDEX). One contains the key value, and the others the data values. The table is searched by incrementing an index variable, and comparing the key variable to the search value. When a match is obtained, the index variable is used to build the complex variable name containing the data.

The Vartable Facility

A *vartable* is an in-storage table with the following attributes:

- A *name*, assigned when the table is created, and used to refer to the table.
- A *scope*, which can be one of:

  PROCESS

  (the default) Indicates that the table is only visible to the NCL PROCESS that allocated it. It is automatically freed when the process terminates, if not explicitly freed beforehand. Table names must be unique within a process, but different processes can operate using different tables with the same name.

REGION

Indicates that the table is visible to all NCL processes in a processing region. For a signed-on user, this means all NCL processes running in either NCL window, under the User Services menu, MAI script procedures, and so on. The table is automatically freed when the region terminates (for example, when the user signs off), if not explicitly freed beforehand. Table names must be unique within the region, but different regions can use different tables of the same name.

SYSTEM

Indicates that the table is visible to all NCL processes in this SOLVE system. The table is automatically deleted when Management Services terminates, if not explicitly freed sometime beforehand.

Note

SYSTEM scope replaced GLOBAL in V3.0 of SOLVE, but the use of GLOBAL is supported.

AOM

Indicates that the statement refers to a mirrored vartable if AOM has been started. For further information see the section, *Mirrored Vartables*, on page 4-15.

- A *keylength*, from 1 to 256 characters long. Each table can have a different keylength.
- Zero or more *entries*, consisting of:
  - A *key.* Every entry must have a unique key, which can be numeric.
  - Up to 16 items of *data,* or an unlimited number of data items if DATA=MAPPED is specified. Each item can be from 0 (null, not present) to 256 characters long. The data value is null if no data is provided for an item when an entry is added.
  - A *counter* field, which is initialized to 0 when an entry is created, and can be reset to any value, or adjusted by any amount, on update calls.
  - A *user correlator field*. This field is maintained by the system, and can be used to provide update synchronization for tables shared between NCL processes (that is, for tables with SCOPE= REGION or SYSTEM only).

The entries in a table are maintained in ascending key field value order. The retrieval option of &VARTABLE allows sequential retrieval in ascending or descending order.

Mirrored Vartables

Note

Mirrored vartables are only available if you are licensed for AOM, or Automation Services products.

A mirrored vartable can be used to control an AOM screening table. For example, message IDs can be stored in a mirrored vartable, and message suppression and routing controled dynamically by adding or deleting entries in the table. Using the LOOKUP screening statement, messages that are matched by message ID can have various attributes assigned or altered.

The ability to dynamically add entries to a mirrored vartable, by screening table code, allows statistics on such things as message ID to be easily accumulated.

If AOM is not started when a vartable with a scope of AOM is allocated, no mirroring takes place. When AOM START is issued, all existing SCOPE=AOM vartables are mirrored. When AOM STOP is issued, all mirrored vartable copies are deleted; the standard vartable remains. If a mirrored vartable is allocated while AOM is started, it is mirrored immediately.

Differences Between Mirrored and Standard Vartables

A mirrored vartable is a standard vartable, allocated with a scope of AOM. SCOPE=AOM is similar to SCOPE=SYSTEM; any NCL procedure can refer to a SCOPE=AOM vartable, but these vartables are logically separate to SCOPE=SYSTEM. That is, there can be a table with an ID=TAB1 in both SCOPE=AOM and SCOPE=SYSTEM.

Mirrored vartables are distinguished from standard vartables by using the &VARTABLE verb to maintain mirrored vartables. This copy can be read from, added to, and updated by an AOM screening table.

In MVS the mirrored copy is maintained in (E)CSA; in VM it is maintained in SOLVE storage. SYSPARMS AOMMIRST=*n* sets the maximum amount of storage that can be used by mirrored vartables.

Updating Mirrored Vartables

When a new entry is added to the mirrored vartable by an NCL procedure, the new entry is automatically copied into the mirrored copy of the table.

The mirrored copy contains:

- A copy of the key (always 16 characters)
- A copy of the first 8 bytes of the DATA1 data field (blank padded if necessary)
- The .AOMID attribute
- An encoded version of the .AOMATTR attribute string
- A counter field .AOMCOUNT

When an existing entry is updated or deleted, the appropriate action is also performed on the mirrored copy. The screening table LOOKUP statement cannot update data in an entry, or delete an entry, but can update the count field in the mirrored entry.

When a screening table LOOKUP statement adds a new entry, the downward mirroring is not performed immediately, but is performed as soon as is necessary. This happens when:

- &VARTABLE GET retrieves an entry with that key value.
- The entry can be sequentially accessed.
- GET OPT=FIRST or LAST is specified. In this instance all pending mirroring is performed immediately.

These rules allow the mirrored copy of the vartable to be accessed with no serialization. This enhances the performance of the screening table.

AOM Attributes of Mirrored Vartables

Mirrored vartables support several extra data fields in a table entry. These are discussed below.

.AOMID

This attribute allows a 1- to 12-character AOM identifier to be stored or retrieved. When a LOOKUP statement assigns the ID attribute on a successful lookup, this value is assigned as the ID of the current message and is available to AOMPROC in the &AOMID system variable. If the supplied value is blank, the AOMID in the table entry is regarded as *not specified*, and a LOOKUP ASSIGN statement of ID will not override the current ID value for the message.

.AOMATTR

This attribute allows you to set a list of AOM message attributes. Each allowable attribute is encoded into a 1- or 2-character value and placed into a particular position in a character string. The blank and dash (-) have special meaning. A blank in any position means that the associated attribute is to be regarded as *not specified*. This means that when a LOOKUP statement assigns an attribute from a table entry that is not specified, the current attribute value is not changed. A dash (-) means that the current value of the associated attribute is not to be changed and acts as a place holder when updating attributes that are further down in the attribute string.

The following message attributes can be encoded in a .AOMATTR string:

Position	Description
1	MVS delete option. Y-YES, N-NO, F-FORCE.
2	ALARM option Y-YES, N-NO.
3	MONITOR option Y-YES, N-NO.
4	INTENSITY option H-HIGH, N-NORMAL, L-LOW.
5	NRD option. Y-YES, N-NO, O-OPER.
6	TRACE option. S-START, F-FINISH, *-This one.
7	ROUTE option. P-PROC, O-PROCONLY, M-MSG, B-BOTH, L-LOG, N-NO. Refer to local/remote route options. If LCLROUTE and/or RMTROUTE are also specified, they override any value set here. If both positions are set and are different, this position is returned as not specified (-).
8	HLITE option. N-NO, R-REVERSE, B-BLINK, U-UNDERSCORE,
9	COLOR option. N-NO, B-BLUE, R-RED, P-PINK, G-GREEN, Y-YELLOW, T-TURQUOISE, W-WHITE.
10-11	MSGCODE value. 2 hexadecimal digits.
12-19	8 user flags. Each can be Y-YES, N-NO.
20	LCLROUTE option. P-PROC, O-PROCONLY, M-MSG, B-BOTH, L-LOG, N-NO

Position	Description
21	RMTROUTE option. P-PROC, O-PROCONLY, M-MSG, B-BOTH, L-LOG, N-NO
22	DOM-TRACK option. Y-YES, N-NO.
23-30	8 RMTCLASS values. Each can be Y-YES, N-NO.

If a string shorter than 30 characters is specified it is padded to 30 characters with dashes. A new entry is regarded as having all attributes as not specified prior to processing any supplied .AOMID or .AOMATTR values.

When retrieving a table entry, if the .AOMATTR values are retrieved, a 30-character string is always returned, however some positions can be blank.

.AOMCOUNT

This attribute cannot be set or updated but can be retrieved by using the &VARTABLE GET statement. This is a counter that is incremented if the COUNT option is specified in the LOOKUP statement. It can be used as a hit counter. A new table entry has this field initialized to zero before the implied count if it is added by a LOOKUP statement with both ADD and COUNT specified.

.AOMTHIT, .AOMTMISS, .AOMTADD

These attributes are counters. The counters can be accessed using these field names in the &VARTABLE QUERY statement.

.AOMTHIT counts the total number of LOOKUP statements, with TOTAL specified, that found a matching table entry.

.AOMTMISS counts the total number of LOOKUP statements, with TOTAL specified, that did not find a matching table entry (including those that later added a new entry).

.AOMTADD counts the total number of LOOKUP statements, with TOTAL specified, that added a new entry. These are also counted in the .AOMTMIS, since no match is found.

These counters are initialized to zero when a table is allocated or reset.

&ZFDBK Values

The &ZFDBK system variable is set by the &VARTABLE verb to indicate the success of the action. Most of the possible values are documented in the *Network Control Language Reference*. Vartables with SCOPE=AOM can also return other &ZFDBK values. These are fully described under each &VARTABLE option in the *Network Control Language Reference*.

&VARTABLE Manipulation Facilities

&VARTABLE provides the following table manipulation facilities:

ADD	Add an entry to a vartable.
ALLOC	Allocate a new vartable.
DELETE	Delete an entry from a vartable.
FREE	Free (unallocate) an existing vartable.
GET	Get (retrieve) an entry from a vartable.
PUT	Add or update (if there) an entry in a vartable.
QUERY	Query the existence of a vartable, and optionally return attribute information.
RESET	Delete all entries in an existing vartable, but preserve the definition of the table (like doing FREE/ALLOC).
UPDATE	Update an existing entry in a vartable.

The functions ALLOC, FREE, RESET, QUERY provide table-level manipulation. ADD, PUT, UPDATE, DELETE, GET are used to manipulate entries in a table.

Shared Table Updating

Tables allocated with a scope of REGION or SYSTEM can be accessed by any number of *concurrently* running NCL procedures. An NCL procedure can be interrupted at any time and Management Services can schedule another procedure. If two procedures are manipulating the same table, the possibility of *logical* corruption exists. The following example illustrates this. (Assume that the entry with key KEY001 has a data content of 10 before the following code is executed.)

```
procedure 1 procedure 2

&K = KEY001
&VARTABLEGET ID=T1 +
            SCOPE=SYSTEM +
            KEY=K FIELDS=D VARS=DATA
&D = &D + 1  -* &D now 11
-*
-* system interrupts procedure 1, schedules procedure 2
-*

                     &K = KEY001
                     &VARTABLEGET ID=T1 +
                                 SCOPE=SYSTEM +
                                 KEY=K FIELDS=D VARS=DATA
                     &D = &D + 1  -* &D now 11
                     &VARTABLEPUT ID=T1 +
                                 SCOPE=SYSTEM +
                                 KEY=K FIELDS=D VARS=DATA
                     &END
-*
-* system re-schedules procedure 1
-*
&VARTABLEPUT ID=T1 +
            SCOPE=SYSTEM +
            KEY=K FIELDS=D VARS=DATA
```

If the data in the table entry was being used as a counter, instead of having 12 (because of 2 adds), it only has 11, as the update done by procedure 2 is lost.

There are three solutions to the above problem:

1. If a single counter is being maintained in each entry, there is no need to get, alter, then update the table entries. Instead, you can use the ADJUST= or FIELDS=(ADJUST) options of the &VARTABLE PUT, UPDATE, or ADD operands to change the system-maintained counter fields in the entry without interference from any other procedure that is using the table. The example above could be written as:

   ```
   &K = KEY001
   &VARTABLEPUT ID=T1 SCOPE=SYSTEM +
              KEY=K ADJUST=1
   ```

   This would insert a new entry if it did not already exist, and add 1 to the (zeroed) counter, *or* update an existing entry by adding 1 to the current counter value.

   This approach is extremely useful for *event* counting, where the event (for example, a particular message ID) is described by the key.

2. Use the &LOCK verb to lock the key value during the manipulation:

```
&LOCK TYPE=EXCL .....  use key as resource name....
&VARTABLE GET ...
...  manipulate the table entry
&VARTABLE UPDATE ...
&LOCK TYPE=FREE .....
```

   The &LOCK verb is described in the *Network Control Language Reference*.

   This approach allows almost any manipulation to take place. However, for heavily accessed tables, the overheads of LOCK/UNLOCK could be excessive. If you do not know the key of an entry in advance, you might need to lock the entire table.

3. If we assume that minimal contention for any one table entry, a third approach is to consider we can access it and only redo our work if someone else actually changes the entry while we are performing calculations.

   Remember that each entry in a vartable contains a *user correlator.* This is a system-maintained update counter. An &VARTABLE GET operation can request that the *current* value of the correlator for the requested entry be returned in a nominated variable. The actual format of the correlator is of no consequence, and should not be altered by the procedure.

   When the procedure issues the &VARTABLE UPDATE, &VARTABLE PUT, or &VARTABLE DELETE, the name of the variable containing the correlator should be supplied in the VARS list (and .USERCORR specified in the FIELDS list). The system checks that the correlator matches the *current* value of the correlator in the *current table entry*, and only if they are the same, performs the update or delete operation (if PUT, and the entry is new, the correlator is ignored).

   Following the update, the correlator in the table entry is given a new, unique value (again, of no consequence to the procedure).

   Obviously, if two procedures issue a GET for the same entry, with no intervening updates to that entry, they will both be given the same correlator value. But, if both then attempt to update that entry, supplying the same correlator value, the second update will fail (with &ZFDBK set to 8), and it should reissue the GET, and redo the update logic.

As an example:

```
&K = KEY001
.RETRY-* loop here if update fails on correlator -*
mismatch

&VARTABLE GET ID=T1 SCOPE=SYSTEM KEY=K +
        FIELDS=(.DATA,.USERCORR) VARS=(D,UC)
           -*
           -* manipulate the entry data (in &D).
           -*
&VARTABLE PUT ID=T1 SCOPE=SYSTEM KEY=K +
        FIELDS=(.DATA,.USERCORR) VARS=(D,UC)

&IF &ZFDBK = 8 &THEN &GOTO .RETRY-* loop if -*
changed
```

Note that, since no locks are being held, there is no extra work to release those locks should the procedure encounter an error condition while manipulating the data.

Also, if the GET is using an OPT= that retrieves a different key value record (for example OPT=GEN), you do not need to know in advance what key value to lock. If &LOCK had to be used in this case, you might need to lock the entire table.

Note

The use of the user correlator can be forced, for a particular table, by specifying USERCORR=YES when allocating it.

(To contrast these two views, (2) can be regarded as the *conservative* approach, and (3) can be regarded as the *aggressive* approach.)

Retrieval Techniques

The &VARTABLE GET statement allows an NCL procedure to retrieve an entry from a vartable, placing the key, data, counter, and possibly user correlator, into nominated NCL variables. The specific entry returned depends on both of the following:

- The value of the OPT= parameter on the &VARTABLE GET statement
- The value contained in the NCL variable nominated on the KEY= parameter of the &VARTABLE GET statement, unless OPT=FIRST or OPT=LAST is specified, in which case the KEY= parameter is not permitted.

The default OPT= parameter is OPT=KEQ, which searches the nominated vartable for an exact EQUAL KEY match.

Most of the other OPT= values are straightforward:

KGE	Retrieve the entry with the lowest key value *greater than* or *equal to* the supplied key value.
KLE	Retrieve the entry with the highest key value *less than* or *equal to* the supplied key value.
KGT	Retrieve the entry with the lowest key value *greater than* the supplied key value.
KLT	Retrieve the entry with the highest key value *less than* the supplied key value.
GEN	Retrieve the entry with the lowest key value *generically equal to* the supplied key value (that is, match for as many non-blank characters in the supplied argument, and the fewest non-blank characters after).

Two OPT= values allow retrieval of the entry with the *lowest* key (OPT=FIRST), or *highest* key (OPT=LAST). When used in conjunction with the OPT=KGT or OPT=KLT options in a loop, a table can be sequentially read (the &VARTABLE GET description illustrates this).

The remaining OPT= value, OPT=IGEN (Inverse GENeric), has special use. The definition is: Retrieve the entry with the longest key value equal to the supplied key value, but at least one character long. For example, a search key of ABCDE searches as follows:

1. For ABCDE—if no match is found, the search continues for...
2. ABCD—if no match, it would then search for...
3. ABC—and so on through...
4. AB and...
5. A

This retrieval option is especially suited for screening messages by message identifier, where generic identifiers (for example, IEF) control all messages that start with that identifier, and less-generic identifiers (degenerating to specifics, such as IEF431I) override the generic identifier in a particular case.

By loading a table with these identifiers, an incoming message identifier can be looked up in one GET OPT=IGEN statement, to find the most-specific match. This avoids looping to examine lists of shorter and shorter identifiers.

&VARTABLE Syntax Descriptions

For a complete description of the &VARTABLE verb and its options see the *Network Control Language Reference.*

5

Arithmetic in NCL

This chapter discusses the following topics:

- About Arithmetic in NCL
- Integer Arithmetic
- Real Number Arithmetic
- Arithmetic Expressions
- Arithmetic Operators
- NCL Substitution and Expressions
- Signed Numbers
- Formatting Numbers

About Arithmetic in NCL

Arithmetic NCL statements yield the mathematical result of a calculation and place it in a nominated variable.

Arithmetic is therefore performed as an *explicit assignment* operation. NCL arithmetic statements are simply a special type of assignment statement.

NCL supports simple or compound arithmetic expressions involving positive or negative numbers. NCL classifies numbers as follows:

- Real numbers that are whole numbers, or numbers containing a decimal fraction. Real numbers can be expressed using scientific notation. The range of numbers supported are those with positive absolute values not higher than +1E+70 and not less than +1E-70, and with negative absolute values not higher than -1E-70 and not less than -1E+70.
- *Integer* whole numbers only, which are a subset of the real number range, but are not expressed in scientific notation, and lie in the range +2147483647 to -2147483648.

Note

Positive and negative real numbers that lie in the range +1E-70 down to -1E-70 are treated as 0.

Integer Arithmetic

NCL performs *integer arithmetic* by default. This means that an expression that contains only integers will yield only integers as a result. For example, the sum:

```
10 / 4 (10 divided by 4)
```

yields the answer 2. The problem with integer arithmetic is that it does not allow you to use decimal calculations to get accurate results. The use of integer arithmetic is controlled by the &CONTROL INTEGER operand.

Real Number Arithmetic

NCL treats numbers with decimal points, or expressed in scientific notation, as *real* numbers and uses floating point arithmetic in the evaluation of real number expressions. The result of a real number calculation, as with integer arithmetic, is placed in a target variable. Unlike integer arithmetic however, the result is not a simple integer but is maintained in the scientific notation form.

The number 22.8 is therefore held in a variable as:

```
+.22800000000000E+02
```

which means:

```
0.228 times 10 to the power of 2
```

A special built-in function, &NUMEDIT, is used to reformat real number results into conventional format so that they can be displayed as standard decimal numbers. &NUMEDIT is described in the section *Formatting Numbers* later in this chapter.

&CONTROL REAL

NCL always uses real number arithmetic if the &CONTROL REAL operand is in effect. This means that even if an operation contains only integers, real number arithmetic is used and the result is placed in the target variable in scientific notation form. It also means that calculations are performed accurately, so that whereas, in the example of integer arithmetic given earlier, the result of dividing 10 by 4 is given as 2, with real arithmetic the answer would be the precise answer of 2.5 but expressed in the result variable as:

```
+.25000000000000E+01
```

If you do not code &CONTROL REAL (which means that the default &CONTROL INTEGER is in force), NCL automatically performs real number arithmetic if real numbers are present in any arithmetic expressions. This automatic switch to real number mode is transparent to your arithmetic functions but is of significance if you wish to perform comparison operations where real numbers are involved.

Comparisons With Real Numbers

You *must* use &CONTROL REAL to switch to real number operation if you wish to perform comparisons of real numbers. This is because the &IF statement does not know implicitly whether a variable containing a value such as:

```
+.98600000000000E+05
```

is a real number in scientific notation or simply a character string that starts with a plus sign.

Arithmetic Expressions

A *simple arithmetic expression* in NCL consists of two numbers (real or integer) separated by an *arithmetic operator*. For example:

```
1 + 2

497021 - 7832

0.08 + 76.889

38 - 97

1E5 - 22.9

44 / 11
```

A number can also be the name of a variable that contains a numeric value, for example:

```
&A + 1

&COUNTER + &INCREMENT
```

The number on the left is operated on by the number on the right. There must be one or more blanks between the operator and the numbers on each side of it.

An expression can also be enclosed in parentheses, in which case blanks are not mandatory between the numbers and the operator. Parentheses can also be used to control the order in which the statement expressions are evaluated.

A *compound arithmetic expression* consists of two or more *numbers and operators, surrounded by parenthesis*, which are processed according to the standard rules of precedence to yield a result, for example:

```
(32 + 6) - (7.8 * 2)
```

would yield `38 - 15.6` giving an answer of:

`+.224000000000000E+02` equivalent to `22.4.`

Arithmetic Operators

The arithmetic operators are symbols used to specify the operation required when a simple expression is evaluated. The operators used by NCL are, in precedence groups:

****	Exponentiation
/	Divide (REAL number arithmetic)
/	Divide quotient (INTEGER arithmetic)
\	Divide remainder (INTEGER arithmetic)
*	Multiplication
-	Subtraction
+	Addition

Note

Within each group, the individual operators have equal precedence. The multiplication operator is therefore no more or less significant than the divide operators. Processing of these operators takes place in strict left to right sequence within the expression.

While most of the operators are familiar, the divide options differ depending upon the arithmetic mode that is being used, REAL or INTEGER, and are explained in more detail below.

Divide (REAL Arithmetic)

When a division operation is performed in which one or both numbers are real, the result is placed in the target variable as a single real number in scientific notation. The result is not split into quotient and remainder.

For example, the expression:

```
&RESULT = (288.5 / 45)
```

yields a value in &RESULT of:

```
+.641111111111110E+01  equivalent to 6.41.
```

Divide Quotient (INTEGER Arithmetic)

When you are using integer arithmetic, the Divide Quotient operator (/) produces division of the left-hand number by the right-hand number and yields the quotient as the result. For example, the answer to:

```
5 / 2
```

is 2.

Note

Even if you have coded &CONTROL INTEGER, where either operand is a non-integer real number, the operation is treated as a real number division and yields a real number result in the target variable.

Divide Remainder (INTEGER Arithmetic)

When you are using integer arithmetic the Divide Remainder operator (\) produces division of the left hand number by the right hand number and yields the remainder as a result. For example, the answer to:

```
5 \ 2
```

is 1.

Note

This operator is invalid for real number arithmetic and causes a syntax error when encountered.

Divide by Zero

An attempt to divide by zero causes a syntax error and terminates the procedure.

Precedence of Operators

In a simple expression there is only one operator so no precedence considerations arise. The calculation is performed according to the operator.

In a compound expression, the evaluation proceeds from left to right with operators being processed according to the standard rules of precedence. So, in a compound expression such as:

(2 + 3 * 4 ** 2)

there are three operators, which are processed in the order:

****	Exponentiation
*	Multiplication
+	Addition

The expression is therefore processed in three steps as follows:

4 ** 2=16

16 * 3=48

48 + 2=50

So an assignment statement coded as:

```
&A = (2 + 3 * 4 ** 2)
```

results in &A being assigned a value of 50.

Using Parentheses to Control Evaluation Order

Although optional, it is often easier to code an expression in parentheses so that the formula being calculated is easier to read. In fact, you might have to code expressions in parentheses to ensure that your calculation is processed as you intend.

For example, the example above shows that the result of the compound expression:

```
(2 + 3 * 4 ** 2)
```

is 50.

However, you might have meant something different and coded:

```
(( 2 + 3 ) * 4 ) ** 2)
```

In this case the expression contained in the deepest pair of parentheses is always evaluated first, which causes the calculation to proceed as follows:

```
( 2 + 3 ) =5
( 5 * 4 ) =20
( 20 ** 2)=400
```

which is obviously a completely different result from the one obtained when the same expression was coded without parentheses.

The use of parentheses to delimit the simple expressions within a compound expression is recommended for clarity of understanding.

NCL Substitution and Expressions

Where a variable is to be used as input to an arithmetic expression, the variable should be enclosed in parentheses to ensure that it is evaluated before an operator is applied to it. This is important where the variable contains a signed number.

For example, consider the following:

```
&A ** 2
```

When &A contains 5, the value is 25. However, if &A contained -5, the answer would be -25. This is due to NCL substitution, which evaluates the expression as follows:

```
-5 ** 2 = -(5 ** 2) = -25
```

To ensure that the sign of the variable value is interpreted correctly, parentheses should be used as follows:

```
(&A) ** 2
```

This will give the correct answer of 25.

Signed Numbers

Positive and negative numbers are supported. The result of an expression which yields a negative number carries a minus sign when assigned to the variable that is the target of the assignment statement. For example:

```
&A = 5 - 8
```

assigns a value of -3 to &A.

```
&A = (288.5 * 2)
```

assigns a value of +.577000000000000E+03 to &A.

Formatting Numbers

When you need to display a number, either integer or real, that is held in a variable, you will often have to format the output so that it is aligned correctly in a field or column of numbers.

The &NUMEDIT built-in function lets you reformat any number held in a variable by specifying three parameters:

- The number of characters that the mantissa (the number to the left of the decimal point) is to occupy, padded on the left with blanks if necessary.
- The number of significant decimal positions that is required.
- Optionally, whether the number is to be kept in exponent format.

Examples of &NUMEDIT usage are as follow. In these examples the character ^ represents a blank.

Expression	**Result**
&A = (2 * 88) &B = &NUMEDIT (0,2,0) &A	(&A = 176) (&B = 176.00)
&A = (2 * 88)	(&A = 176)
&B = &NUMEDIT (5,2,0) &A	(&B = ^^176.00)
&A = (4.8 + 6.998)	(&A = +.117980000000000E+02)
&B = &NUMEDIT (8,5,) &A	(&B = ^^^^^11.79800)
&A = (2.3 ** 2)	(&A = +.529000000000000E+01)
&B = &NUMEDIT (4,4,E) &A	(&B = ^^^5.2900E+00)

See the &NUMEDIT verb description in the *Network Control Language Reference* for further information.

Note

The width of the mantissa will always be at least the number of characters in the mantissa, even if the mantissa parameter is less than the actual number of characters. The first example shows this, where the mantissa parameter is 0 but the mantissa length is three characters (176).

The E format of &NUMEDIT always returns one decimal digit to the left of the decimal point.

6

Designing Interactive Panels (Panel Services)

This chapter discusses the following topics:

- About Panel Services
- Designing Panels
- Analyzing Panel Input
- Monitoring Panel Return Codes
- Handling Errors
- Finding Out Which Input Fields Have Changed
- Output Padding and Justification
- Input Padding and Justification
- Processing with Light Pens/Cursor Select
- Intercepting Function Keys
- Using Panels on Different Screen Sizes
- Controlling Cursor Positioning
- Dynamically Altering Panel Designs (PREPARSE)
- Controlling the Formatting of Input Fields
- Retrieving Panels from Panel Libraries
- Displaying Panels on OCS Windows
- Using Asynchronous Panels

About Panel Services

The Panel Services facility of Management Services lets you design and implement your own Non Programmable Terminal (NPT) full-screen display formats. Using this facility, you can create NCL procedures for communicating with terminals using full-screen displays, referred to as *panels* throughout this chapter. These panels enable NCL processes to display output data for you, and can be designed with input fields for communicating back to these NCL processes:

- Panel Services can be used by NCL processes executing in any NCL environment associated with a display window.
- Panel Services is supported by an online editor which can be used to create and modify panels.
- Panels can be defined to take advantage of such 3270 features as light-pen and cursor selection. Full-color and extended highlighting support is available on the appropriate terminals.
- Internal editing and validation can be selected for individual input fields to minimize the validation required within NCL procedures.

Logical Screen Manager (LSM)

Display panels are monitored by a Logical Screen Manager (LSM) which means that only changed data is written to the screen. This ensures that the minimum amount of necessary data is transmitted. Wherever possible, LSM compresses datastreams to ensure they operate efficiently. Using an LSM provides other benefits:

- The *peppering* effect of data as it is written to some displays is eliminated.
- Screen flashing is minimized.

Creating or Changing Panels

You create and change panels by using an on-line editor—select option P from the MODS : Primary Menu. You must be authorized to use this facility. Sites can limit the number of concurrent users by restricting the amount of storage available to the on-line editor. See the chapter titled *Maintaining Panels* in your *MODS Programming and Administration Guide* for detailed information on using the on-line editor.

When you create a panel, you must give it a unique 1- to 12-character name, which is nominated whenever a request is made to display the panel. Once defined, panel details are stored in a VSAM database and can be updated as required.

The Management Services standard split-screen facilities are very useful when designing panels: the panel editor can be used on one window, while the panel test facility of Edit Services displays the current version of the panel being developed on the other.

Invoking Full-screen Panels

Once defined and saved, a panel can be displayed immediately. Panels for display are usually selected using an NCL procedure containing an &PANEL statement. Other NCL statements such as &LOGON can nominate a panel for display under specific circumstances.

If you try to display a panel that has not been defined, the procedure terminates and displays an error message. You can use the &CONTROL PANELRC statement to receive notification of processing anomalies (including referencing non-existent panels). Your procedure can detect this fact and pre-empt termination to continue processing.

Synchronous and Asynchronous Panel Displays

An NCL procedure can issue an *asynchronous* panel statement or a *synchronous* panel statement:

- Asynchronous panel statements provide a panel display, but the procedure continues execution without waiting for input from the terminal.
- If a synchronous panel is displayed, further processing of the NCL procedure stops until you press an ENTER key, a Function key, or provide some other input from a light pen or cursor select key. Control then returns to the NCL procedure following the &PANEL statement. The NCL procedure can then process the data just entered from the terminal.

If a synchronous panel has been displayed for a specific length of time without input from an operator, control can be returned to an NCL procedure automatically. The default panel operation is synchronous.

Note

The synchronous state is simpler to understand than asynchronous, and descriptions of panel operations in this chapter assume synchronous operation unless stated otherwise.

Asynchronous panel statements are detailed in the section on *Using Asynchronous Panels*, on page 6-40.

Fixed and Variable Data in Panels

Panels contain a combination of fixed data and variable output data:

- Fixed data is the screen captions, field identification text, and other static screen information defined when the panel is created. This does not change when the panel is displayed.
- Variable output data is data generated by the system when the panel is displayed. It replaces variables positioned within the panel created by the editor. Data is extracted from NCL variables available at the time the panel is invoked.

Variable output data can be displayed in one of the following:

- Protected output-only fields (where the data comprises either system or user variables)
- Unprotected input fields, for any user variables. Once displayed, you can enter data into the unprotected input fields. Panel Services then inserts this data into the user variable for each field, so it is available for further processing by NCL procedures.

The syntax for defining variables within a panel is discussed later in this chapter.

Designing Panels

Each panel design can contain a series of up to 62 lines, each 80 characters long, and one or more fields. Each field is preceded by a field character which identifies the attributes for that field. These attributes prescribe the following:

- The field type (input, output, selector pen detectable, or null)
- The intensity (brightness) of the display
- Optional editing rules
- The color and extended highlighting used when the field is displayed (for appropriate terminals)

Design Guidelines

When creating or modifying a panel, enter panel details exactly as you want them to display. To avoid confusing the users of your panel:

- Give careful consideration to the use of highlighting and color. Ensure these are not used excessively and do not detract from their effectiveness
- Use color selection and message placements *consistently* for all panels in a similar series
- Clearly identify and caption input fields to describe the input data you require

- Do not have too many fields on a panel. If necessary, spread data or solicit input over several panels to avoid crowding a single panel.

When a panel is to be displayed, its associated control statements are parsed and any variable substitution performed. This allows control statements to be dynamically tailored.

Field Characters

Each panel line has one or more *fields*, starting with a *field character* which prescribes the field attributes.

You must specify a #FLD control statement for each field character your panel requires. There are two ways of defining field characters:

- *Character mode*. To specify character mode, use any special character other than an alpha or numeric character, and excluding ampersand (&), blank, or null.
- *Hexadecimal mode*. To specify hexadecimal mode, enter the hex value for the character (for example, X‘FA’). Use any hex value in the range X‘01’ to X‘FF’, excluding the values X‘0E’, X‘0F’ and any values which correspond to the EBCDIC values of characters not allowed on character mode fields. Hexadecimal mode is used where you need a very large number of field types within one panel and there are insufficient special keyboard characters available to accommodate all of the field characters you require.

Note

If using hexadecimal mode field characters, you must prime the panel definition with the preparse option to assign correct values to field character positions in the actual panel.

Field Types

Each field is allocated a *field type* which prescribes the method for processing the field. Four field types are supported:

OUTPUT	Display only - no data can be entered from the screen.
INPUT	You can both display and enter data.
SPD	Selector pen detectable - data cannot be typed in.
NULL	Display only. Although unprotected, any data entered will be ignored.

Any mixture of the above field types can be defined to suit the requirements for a panel you are designing.

The field character that precedes each field determines:

- The field type
- The display characteristics of the field (intensity, color, highlighting)
- (For input fields) the internal validation rules that must be obeyed for data entered in that field. Such rules can specify, for example, that a field is mandatory, must be numeric, cannot contain imbedded blanks, or must be a valid date, and so on.

Each field character you define, occupies the equivalent screen position when the panel is displayed, but appears as a blank character (the *attribute byte*).

The field proper starts from the next position after the field character, and continues to the next field position on the same line, or to the end of that line where there is no intervening field. Fields do not wrap round from one line to the next.

The three standard default field characters are:

%	high-intensity, protected (no input)
+	low-intensity, protected (no input)
_	high-intensity, unprotected (input, no validation).

These do not require definition by a #FLD statement.

Define any additional field characters you need using the #FLD statement. The attributes for the defaulted characters above can be modified. The #OPT statement can be used to nominate alternative standard field characters, so that %, +, and _ can be used within the panel and not processed as field characters.

Column 1 of each line of a panel must be a valid field character; if one is not defined, the attributes for the second standard field character, (normally as +, for low-intensity, protected) are used to replace any data incorrectly placed in that column.

Figure 6-1. Sample Panel Using Default Field Characters

```
%-------------------- Electronic Memo --------------------------
+SELECT  OPTION%===>_SELECT+
%       1+Create a Memo
%       2+Send a memo
%       3+Display incoming memo
%       4+Delete an incoming memo
%       5+Exit from this function
```

In Figure 6-1, all fields preceded by % display in high-intensity and are protected from data entry. All fields preceded by + display in low-intensity and are also protected. The only field available for input is on the third line, preceded by an underscore (_). The word SELECT identifies the NCL user variable that receives the data you enter in this field once the ENTER key is pressed.

By default, the cursor is placed at the SELECT field, as this is the first (and only) field requiring input—no other cursor position has been specified.

Note

The ampersand (&) normally associated with a variable is omitted here.

Assume that our sample panel, as defined in Figure 6-1, is called PANEL1. The NCL procedure to display this panel is:

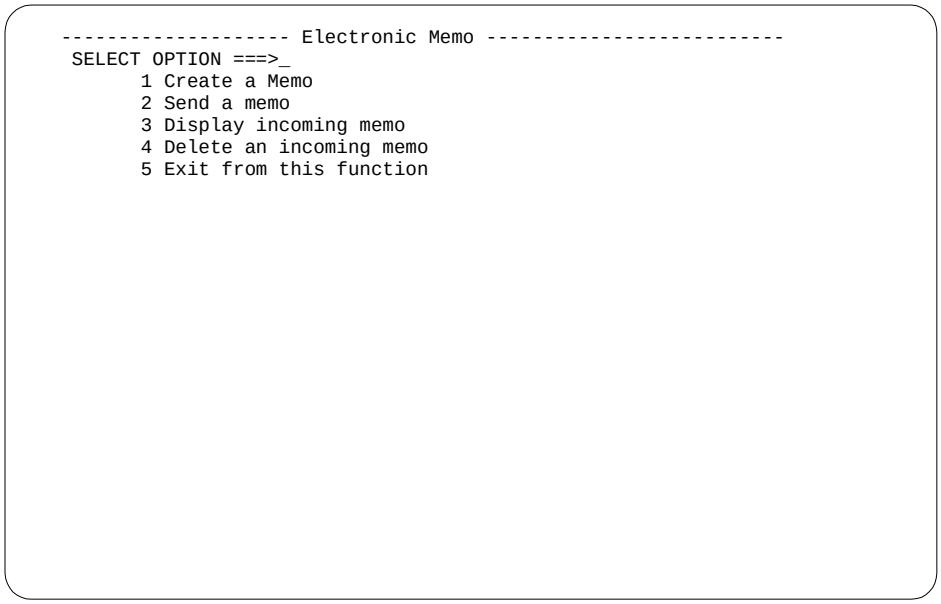
```
&PANEL PANEL1
```

When displayed, field characters are removed and the required terminal attribute characters substituted. Figure 6-2 shows how the panel defined in Figure 6-1 appears, when displayed.

Note

In all figures, the underline symbol (_) designates the cursor location.

Figure 6-2. Sample Panel Display



Overriding the Input Attribute

Panels generally contain a set of input and output fields which remain relatively fixed. The fields might need to be switched from input to output as a group.

For example, a panel used to add, browse, and update a record has three groups of input fields. They can be:

- Only input during add
- Input during an update (data fields)
- Always input (command fields and so on.)

Separate field attribute characters can be used and switched from TYPE=INPUT to TYPE=OUTVAR.

There are however special cases (field level security for example) where individual fields or unpredictable groups of fields might need to be protected. The &ASSIGN OPT=SETOUT verb can be used to force the protection of a variable, even if it appears in an input field in a panel. A check is made against the current standard field attributes when a field is forced to output. If the attributes of the field match the attributes of one of the standard input fields, the field attributes (such as color and highlighting) are switched to the attributes of the corresponding output field.

Controlling How Long a Panel is Displayed

Panels do not always have to accept operator input, although this is the most common mode of operation. NCL interfaces to other system components allow you to set up monitoring functions with display-only panels requiring little or no operator entry.

Alternatively, you might want an NCL procedure to resume control if no input is received within a certain time.

By default, whenever a synchronous panel is displayed, Panel Services waits indefinitely for entry from an operator. However, the INWAIT operand of the #OPT control statement lets you override this.

INWAIT lets you specify a time (in seconds or parts of seconds) for Panel Services to wait before control is returned to the invoking NCL procedure. During this interval, standard keyboard entry is accepted and processed normally. Once the time interval expires, control returns to the invoking NCL procedure.

This facility has many applications. For example:

- If a panel displays many high-intensity fields for long periods, you can get screen burnout. Use INWAIT to force a return of control after 20 minutes (for example), when a blank panel replaces the display panel. The original panel can be redisplayed when any input is received.
- If you specify INWAIT=0, keyboard entry is not accepted and control returns to the NCL procedure once the panel is displayed. This could be useful for displaying a message while the NCL processes the last user request.

See the #OPT statement description in the reference section for detailed information on the format of the INWAIT operand.

Note

The INWAIT function does not apply to asynchronous panel operations.

Analyzing Panel Input

The &INKEY system variable returned to the NCL procedure following a synchronous &PANEL statement, is set to indicate how input was made (for example, by using the ENTER key).

After displaying a panel, Panel Services waits either for you to complete input (signalled by pressing the ENTER key or some other function key), or for a time-out interval to expire (where an INWAIT period has been specified on the #OPT control statement).

The &INKEY system variable can be tested to find out whether operator input has occurred, and to provide support for Function keys.

If the INWAIT time period elapses, &INKEY is set to a null value. If INWAIT did not expire, &INKEY is set to one of the following strings:

Entry type	**&INKEY value**
ENTER key	ENTER
Function key	F01 to F24 (always 4 characters)
PA key	PA1 to PA3
LIGHT PEN	ENTER

Note

If &CONTROL PFKMAP is in effect, F13 to F24 are presented as F01 to F12.

&INKEY can be tested like any other system variable, for example:

```
&IF &INKEY EQ PF01 &THEN &PANEL HELP
```

Remember that where INWAIT is used a null value can be set for &INKEY. Therefore, &IF statements using &INKEY must allow for a possible null value syntax error if &INKEY is eliminated from the statement by variable substitution. This can be achieved as follows:

```
&IF  .&INKEY  EQ  .  &THEN &GOTO .NOINPUT
```

Alternatively, to determine if the INWAIT timer has expired, use return codes from Panel Services as requested by &CONTROL PANELRC. In this case, a return code of 12 is set in the &RETCODE system variable to indicate the INWAIT time interval has expired.

The section on the &GOTO verb in the *Network Control Language Reference* describes how you can greatly simplify testing individual Function key values, using direct branching techniques.

For example:

```
&CONTROL NOLABEL
.DISPLAY
           &PANEL MYMENU
           &GOTO .MENU&INKEY
           .
           .
           drops through if key not supported.
           .
           .
           &SYSMSG = &STR INVALID SELECTION
           &GOTO .DISPLAY
           .MENUENTER-* comes here if ENTER key pressed
                 .
                 .
           .MENUPF01-* comes here if F1 pressed
                 .
                 .
           .MENUPF02-* comes here if F2 pressed
                 .
                 .
           ....
```

The &GOTO .MENU&INKEY statement is resolved as an &GOTO to label constructed by the current value of &INKEY suffixed to .MENU. If the label is not found, the &GOTO acts as a null statement. Control passes to the next statement because the &CONTROL NOLABEL operand was used.

Note

&INKEY is a system variable, so any attempt to assign a value into &INKEY results in an error.

Monitoring Panel Return Codes

An NCL procedure can enhance the available synchronous panel processing options by issuing an &CONTROL PANELRC statement before displaying a panel. If this option is used, then:

- After panel processing has completed (that is, following execution of an &PANEL statement) the &RETCODE variable is set to indicate a particular processing condition and control passes to the NCL statement immediately following the &PANEL statement.

- The NCL procedure is responsible for testing the value of &RETCODE immediately after the associated &PANEL statement and processing accordingly.

Note

Failure to test &RETCODE can result in unexpected results.

Asynchronous &PANEL statements always set an &RETCODE completion code, even if &CONTROL PANELRC has not been issued.

The return codes 0, 4, and 8 are set as a result of a process called *internal validation*, which automatically edits input fields according to rules prescribed in the panel definition. The internal validation process is described fully in subsequent sections.

&RETCODE = 0

For *synchronous* panel operations: One or more panel input fields have been modified. This return code indicates if any field data have been changed. You could use this to decide if any further validation is required. If data has been overtyped but not altered, it is not considered to have been modified. If internal validation detects an error, return code 8 is set.

Note

If &CONTROL FLDCTL is set and any input field on the panel has MODIFIED attribute, the panel returns &RETCODE 0, even if the user did not physically change the field.

For *asynchronous* panel operations: The panel has not been updated but input has been received for an earlier display of the same panel. In this case, you can now process the input, but you must reissue the &PANEL statement to update the panel display.

&RETCODE = 4

The same as for return code 0, except that no input fields on the panel have been modified. This return code can be used with return code 0 to decide whether further data processing is required.

Warning

Take care when using this return code if multiple interactions with the panel can occur.

If internal validation detects an error, return code 8 is set. Return codes 0 and 4 apply only to the last input from the screen; if the procedure accepts input from a screen where some data is changed and re-displays the panel, which is then not modified, any earlier indications of data modification are lost. Your procedures must allow for this situation.

This return code can be produced for both synchronous and asynchronous panel operations.

&RETCODE = 8

An internal validation error has been detected. When &CONTROL PANELRC is in effect, automatic error condition processing is suppressed so the procedure is notified by this return code. The &SYSMSG variable contains the text of the error message (for example, NOT WITHIN RANGE). The &SYSFLD variable contains the name of the input field in error—this is the name of the variable (minus the & prefix) which receives data entered into that field. When this return code is set, the procedure can ignore the error and take alternative action. For example:

- Displaying a Help panel
- Altering the error message text by assigning a new message to the &SYSMSG variable
- Redisplaying the panel by having the #OPT ERRFLD facility display the error message unchanged and putting the cursor on the field in error. In this case, the ERRFLD operand can be defined on the #OPT statement as ERRFLD=&SYSFLD to simplify processing.

Note

Using &RETCODE = 8 is an ideal way of providing an escape mechanism (such as F3), even though the panel has been defined as having mandatory fields REQUIRED=YES. (See the section titled *Internal Validation*, later in this chapter.)

&RETCODE = 12

For *synchronous* panel operations: The panel display time-out limit specified by the #OPT INWAIT operand has elapsed without any data entry. The &INKEY system variable is null. (The INWAIT option is discussed in the section, *Controlling How Long a Panel is Displayed*, on page 6-9.

For *asynchronous* panel operations: This return code means that the panel has been displayed and has returned to the issuing procedure. Successive updates of the panel can be made by repeating the &PANEL statement; &RETCODE = 12 indicates the panel will be redisplayed and that no input has been received from earlier displays of the same panel. (The availability of input from a previous display of the panel is indicated by return codes 0 or 4, as described above.)

&RETCODE = 16

The panel requested is not defined in the current panel library path, or else some other serious error has occurred which prevents the panel displaying. The &SYSMSG system variable contains a message describing the error condition.

If &CONTROL PANELRC is not in effect, the following processing occurs:

- If required, the NCL procedure determines for itself whether any input fields have been modified. See the section, *Finding Out Which Input Fields Have Changed*, on page 6-21.

- Internal validation handles all error processing automatically and re-displays the panel if errors occur.
- The NCL procedure can only determine whether the INWAIT interval has expired by testing the &INKEY variable for a null value.
- If a requested panel does not exist in the current panel library path, or some other serious error occurs, the NCL procedure terminates with an error message.

Handling Errors

NCL procedures that use Panel Services usually contain processing to validate operator input. If an error is detected, the procedure must be able to notify you about the field in error and the nature of the error.

Errors can be identified by taking advantage of the variable substitution performed for panel control statements, setting appropriate variables for the #OPT control statement CURSOR and ALARM operands, moving suitable text into an error message variable, and redisplaying the panel.

This technique works well, but places additional responsibility on your NCL procedure to ensure that the correct variables are set and cleared later if no errors are detected. (The situation is compounded as the number of fields on the panel increases.)

It also helps the end-user if you highlight those fields in error by switching them to another color, or by using facilities such as blinking or reverse-video on terminals that support these facilities. This can also be achieved by substituting variables in control statements, but becomes unwieldy.

Panel Services facilities are provided to simplify the notification of errors. These facilities assume that error reporting includes additional attention-getting operations such as:

- Ringing the terminal alarm
- Placing the cursor on the field in error
- Displaying error message text
- Changing the field in error to high-intensity
- Switching the field in error to a different color
- Displaying the field in error in reverse-video

The #ERR control statement lets you design a common environment (that is, a common set of attributes) for error conditions. This environment is then applied

whenever an input field is nominated as being in error by one of the following methods:

- Using the &ASSIGN verb. (The syntax and usage of the &ASSIGN verb is described in the *Network Control Language Reference*. Note especially the OPT=SETERR and OPT=RESETERR options.) This method lets you set the error attribute for multiple fields
- By nominating the field name in the ERRFLD operand of the panel services #OPT statement. This method allows the error attribute to be set for a single field. The ERRFLD operand normally specifies a variable (for example, ERRFLD=&SYSFLD). When the NCL procedure wants to mark a field as being in error, it places the name of the field in error into the variable (for example, &SYSFLD = FIELD1).

Figure 6-3. Panel Using ERRFLD Operand

```
#OPT ERRFLD=&SYSFLD
#ERR ALARM=YES INTENS=HIGH COLOR=RED HLIGHT=REVERSE
%--------------------  Name and Address  ----------------------------
%&SYSMSG
+ENTER NAME%===_NAME                     +
+ENTER ADDR%===_ADDR1                    +
+ENTER ADDR%===_ADDR2                    +
+ENTER ADDR%===_ADDR3                    +
+ENTER ADDR%===_ADDR4                    +
+ENTER ADDR%===_ADDR5                    +
```

The example in Figure 6-3 illustrates the use of the #OPT ERRFLD method. Note that this method requires two items in panel control statements: a #OPT ERRFLD statement, and a #ERR statement.

Assume the ADDR5 field data must be XYZ; initially the panel is displayed with the &SYSFLD variable set to null. Because the ERRFLD operand does not nominate a field, the #ERR statement is ignored and the panel is displayed normally.

On subsequent entry, the validation procedure checks the ADDR5 field and determines that it is wrong. The &SYSFLD variable is set to the name of the incorrect field (ADDR5), the appropriate error text is assigned to the &SYSMSG variable, and the panel is redisplayed.

```
.DISPLAY
   &PANEL MYPANEL

   &IF .&ADDR5 NE .XYZ &THEN &GOTO  .ERROR
   .
   .  other processing
   .
.ERROR
   &SYSFLD = ADDR5
   &SYSMSG = &STR DATA MUST BE XYZ, RE-ENTER
   &GOTO .DISPLAY
```

When the panel is redisplayed, the cursor is positioned on the last address field, ADDR5. This field is displayed in high-intensity and the terminal alarm rings (on a color terminal, the field is displayed in reverse-video and red.)

Note

To achieve the same result using &ASSIGN verb instead, remove the #OPT statement from the panel definition and replace the line:

```
&SYSFLD = ADDR5
```

with

```
&ASSIGN OPT=SETERR VARS=ADDR5
```

This method lets you use the #ERR attributes for more than one field.

If the ADDR5 field is corrected and the panel redisplayed, the ADDR5 field returns to its original attributes.

If the erroneous field is a non-display field (such as those used to enter passwords), the overriding attributes that force the entered data to be displayed are ignored and the field remains a non-displayed field.

Note

The variable &SYSFLD is a special variable which is cleared by Panel Services after the panel is displayed (see the section, *Advanced Internal Validation*, on page 6-19). If you use another variable such as &ERROR, you should clear it after all error conditions have been corrected. This ensures that if another panel that uses the same variable is displayed, it does not incorrectly signal an error. The &SYSMSG variable need not be cleared by the NCL procedure as it is always reset to null by Panel Services before returning to the NCL procedure (see the next section, *Internal Validation*).

Internal Validation

In your own NCL procedure you can perform all necessary panel field validations. However, Panel Services includes powerful automatic editing capabilities for this function, called *internal validation*.

The Electronic Memo panel definition shown in Figure 6-1 does not specify any Panel Services validation for data entered in the SELECT field—you need to define an NCL procedure to validate input and redisplay the panel with an error message.

You can do this by using the following code:

```
&CONTROL NOENDMSG
    .PANEL
       &PANEL PANEL1
       &IF .&SELECT EQ .  &THEN &GOTO .ERR1
       &A = &TYPECHK (NUM) &SELECT
       &IF .&A NE .NUM &THEN &GOTO .ERR2
       &IF &SELECT GT 5 OR +
          &SELECT LT 1 &THEN &GOTO .ERR2
       .
       .  other processing
       .
       .ERR1
          .
       .  build error message and then redisplay the panel.
          .
       .ERR2
          .
       .  build error message and then redisplay the panel.
          .
```

Panel Services can provide internal validation at a field level. This can perform most of the basic editing required for a field and can greatly simplify the processing required within an NCL procedure.

The type of field validation you require can be specified on the #FLD statement for the field character attributes marking the start of a field. Multiple #FLD control statements can be used to specify different validation criteria for individual fields.

See the #FLD statement description, on page 6-52, for details on the types of validation possible.

Figure 6-4. Panel Using Internal Validation

```
#FLD _ REQUIRED=YES BLANKS=TRAIL RANGE=(1,5)
%-------------------  Electronic Memo  ----------------------------
+SELECT OPTION%===>_SELECT +
%&SYSMSG
%    1+Create a memo
%    2+Send a Memo
%    3+Display incoming memos
%    4+Display incoming memo
%    5+Exit from this function
```

Figure 6-4 shows a panel which includes the #FLD statement to define validation requirements.

In this example the standard underscore character (_) has been redefined:

- To specify internal validation for ensuring the field is not omitted (REQUIRED=YES)
- So that trailing blanks but not imbedded blanks can be accepted (BLANKS=TRAIL)
- So that the entered data must be numeric and in a range from 1 to 5 (RANGE=(1,5))

Internal validation error processing depends on how the &CONTROL PANELRC option is set. If &CONTROL PANELRC is not in effect, *automatic internal validation* occurs; if it is in effect, *advanced internal validation* can be used.

Automatic Internal Validation

An input error detected by internal validation where &CONTROL PANELRC is *not* in effect, is automatically handled by Panel Services. Control is not returned to the NCL procedure. (The next section, *Advanced Internal Validation*, explains what happens when &CONTROL PANELRC *is* in effect.)

When an error is detected by internal validation, Panel Services assigns the appropriate error text to the &SYSMSG variable, automatically redisplays the panel, rings the terminal alarm, and places the cursor at the field in error. The display options defined within the #ERR statement for the panel are applied to the erroneous field (for example, color or highlighting attributes).

When designing your panel, ensure the &SYSMSG variable field is described somewhere on it:

- Position the &SYSMSG field towards the top of the panel so any message it includes is visible in split screen mode. If &SYSMSG is not provided, the error text cannot display and the operator might not be able to find the cause of error. (The error message texts used by internal validation are detailed in the #FLD control statement EDIT operand.)
- The field for the &SYSMSG variable is normally a high-intensity field, or some prominent color such as red (for color terminals).
- &SYSMSG always resets to a null value when control returns to the NCL procedure if no internal validation is performed and no errors are detected. This ensures an NCL procedure does not have to explicitly clear the &SYSMSG variable if no error is found.

The &SYSMSG variable is not limited to internal validation uses. NCL procedures can use it to display error messages or text assigned during processing.

Internal validation facilities cannot address all of the validation requirements a procedure needs to perform. Like other user variables, the &SYSMSG variable does not span different nesting levels—it is unique to each level unless made available using the &CONTROL SHRVARS option.

Note

Assigning multiple word error text to the &SYSMSG variable requires that you use the NCL &STR or &ASISTR function. For example:

```
&SYSMSG = &STR INVALID SERIAL NUMBER
&SYSMSG = &ASISTR RESOURCE NOT ACTIVE
```

Advanced Internal Validation

While the automatic internal validation greatly simplifies an NCL procedure, sometimes you might want a procedure to continue processing when a validation error is detected, bypassing the automatic reshow normally performed. For example, if a field is specified as mandatory (REQ=YES), automatic internal validation forces the field to be entered and issues this message:

```
REQUIRED FIELD OMITTED
```

Alternatively, you might want a system where a field is mandatory unless F1 is pressed to display a Help panel, or where F12 indicates the entry is to be bypassed. You might also want to change the text of the messages supplied by internal validation.

An &CONTROL PANELRC statement issued before an &PANEL statement tells Panel Services processing that the NCL procedure is designed to cater for a range of &PANEL return codes. These return codes accommodate a variety of conditions possible when processing a panel, and signify there is no automatic reshow performed after an error is detected.

If &CONTROL PANELRC is used, the procedure receives control if internal validation detects that a required field has been omitted (or some other error). The name of the field in error is supplied in a variable called &SYSFLD and the text of the error message registered by internal validation is supplied in the &SYSMSG variable.

In the example above, where F1 and F12 require special processing, the procedure tests &INKEY for PF01 or PF12, ignores the error, and reacts as required. If F1 or F12 is not pressed, the procedure redisplays the panel with the error message.

When redisplaying the panel, the &SYSFLD variable containing the name of the field in error can be referenced on the #OPT statement ERRFLD operand (ERRFLD=&SYSFLD). This initiates the processing which positions the cursor on the field in error (and possibly a #ERR statement designating special attributes to be applied to the field in error). The text in the &SYSMSG variable can be modified if required, or assigned into another variable.

If a procedure uses this technique, it must be written to handle all return codes possible from the &PANEL statement.

If no internal validation is performed or no internal validation error is detected, &SYSMSG and &SYSFLD are set to a null value when control is returned to the NCL procedure after the &PANEL statement. This ensures that the NCL procedure does not have to explicitly clear variables if an error is not found.

Like &SYSMSG, the &SYSFLD variable is not limited to internal validation uses. It can also be used by the NCL procedure for its own error indications:

- Once a procedure has detected an error it can assign error message text into the &SYSMSG variable, place the name of the field in error in &SYSFLD, and redisplay the panel.
- If the panel uses the ERRFLD option on the #OPT statement, the error message is displayed and the cursor positioned at the field in question. Any other error processing defined on the #ERR statement is also performed.

Like other user variables, the &SYSFLD variable does not span different NCL procedure nesting levels and is unique to each level unless shared under &CONTROL SHRVARS. Regardless of other uses, &SYSFLD resets to null after the &PANEL verb unless an error is detected by internal validation.

Finding Out Which Input Fields Have Changed

NCL procedures need a simple mechanism for finding out which input fields have been modified on a display panel with multiple fields. This lets you validate only those fields which were changed.

When an NCL procedure executes with the &CONTROL FLDCTL option set, Panel Services processing automatically creates a *stack* of all modified input fields returned from the terminal. This stack is built by scanning the input panel line by line from top to bottom, and from left to right. The system variable &ZMODFLD is primed with the name of the input field variable that is logically at the top of the stack. Each time the &ZMODFLD variable is referenced, its value changes to the name of the variable associated with the next modified input field on the panel. When the procedure processes the last panel input field variable name, &ZMODFLD is reset to a null value.

For example:

```
&CONTROL NOLABEL
   .
   .
   .
&PANEL GETABC      -* Display panel containing input fields
                   -* &A, &B, and &C.
   .
   .               -* User enters fields A and C.
   .
   .               -* Procedure resumes processing
   .               -* &ZMODFLD = A
   .
.INPUTLOOP
   &GOTO .&ZMODFLD -* Process next modified field -*
variable
   &GOTO .NEXTPANEL -* No more fields...issue next panel.
.A ...             -* Process input variable A.  This is
                   -* the first reference to &ZMODFLD,
                   -* whose value now changes to the next
                   -* variable name on the stack, that is, C.
&GOTO .INPUTLOOP
.C ...             -* Process input variable C.  This is
                   -* the second reference to &ZMODFLD,
                   -* whose value now becomes null, since C
                   -* is the last modified field variable.
&GOTO .INPUTLOOP
```

An NCL process can have one active &ZMODFLD stack at a time. If another panel is displayed while the &CONTROL FLDCTL option is still in force, then the current &ZMODFLD stack is rebuilt. &ZMODFLD variables that are not accessed remain in the stack if they are in the panel just displayed.

Alternatively, if &CONTROL NOFLDCTL is issued to suspend the &ZMODFLD stack generation, any number of other panels can be presented without destroying the &ZMODFLD stack. The &ZMODFLD stack is available for use unchanged as soon as &CONTROL FLDCTL is reissued. Certain options of the &ASSIGN verb help manipulate the &ZMODFLD stack (see &ASSIGN verb description in the *Network Control Language Reference*).

Output Padding and Justification

Careful use of padding and justification greatly enhances the usability of panels for end-users. Panel Services includes extensive facilities to manipulate displayed data. Padding and justification qualities are specified by the #FLD statement. There are two justification categories—field level justification and variable level justification. Both can be used concurrently.

Field Level Justification

This is performed on an entire field as delimited by defined field characters. Field justification analyzes the entire field, strips trailing blanks, and pads and justifies the remaining data. The #FLD operands controlling field level justification are JUST and PAD.

The various ways data can be manipulated are best described by a series of examples. These examples show a mix of fields each defined with a different field character and each showing a different display format. Study the #FLD statements and observe the results achieved.

```
#NOTE This sample panel definition gives examples of the
#NOTE use of field level justification and padding.
#FLD#
#FLD$ JUST=RIGHT
#FLD@ JUST=LEFT PAD=<
#FLD? JUST=RIGHT PAD=>
#FLD/ JUST=CENTER PAD=.
#&VAR01                          +
$&VAR02                          +
@&VAR03                          +
?&VAR04                          +
/&VAR05                          +
```

Assume the following variable assignment statements are executed by the NCL procedure before displaying the sample panel:

```
&VAR01 = &STR Left justified null padding
&VAR02 = &STR Right justified null padding
&VAR03 = &STR Left justified with padding
&VAR04 = &STR Right justified with padding
&VAR05 = &STR Center justified with padding
```

The default values are JUST=LEFT and PAD=NULL, as shown by the first line in the example below, where field character # is used with no attributes other than the defaults. The sample panel is displayed as follows:

```
Left justified null padding
                                Right justified null padding
Left justified with padding<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>Right justified with padding
..............Center justified with padding..............
```

Variable Level Justification

This operates independently of field level justification, and applies to the data substituted for field variables defined as requiring variable level justification. Variable level justification is designed to help tabulated output, where data of differing lengths is substituted for a series of variables and where the normal substitution process would disrupt display formats. The #FLD operands that control field level justification are VALIGN and PAD.

The substitution process normally substitutes data in place of the &variable without creating additional characters. Thus, if a variable (for example, &VARIABLE) is replaced by data, any characters following this are moved left to occupy any spaces remaining after substitution (that is, if spaces are freed going from a long variable name to a shorter data length).

```
#NOTE This sample panel definition gives examples of the
#NOTE use of variable justification, padding, and field
#NOTE justification.
#FLD # VALIGN=LEFT
#FLD $ VALIGN=RIGHT
#FLD @ VALIGN=CENTER
#FLD ? VALIGN=LEFT PAD=.
#FLD / VALIGN=RIGHT PAD=.
#FLD } VALIGN=CENTER PAD=.
#FLD ! VALIGN=LEFT JUST=RIGHT PAD=.
#&VARIABLE other data                +
$&VARIABLE other data                +
@&VARIABLE other data                +
?&VARIABLE other data                +
/&VARIABLE other data                +
}&VARIABLE other data                +
!&VARIABLE other data                +
```

Variable level justification controlled by the VALIGN operand of the #FLD statement, lets you influence the way substitution is performed.

Note

Variable level justification is only performed if the length of the data being substituted is less than the length of the variable name being replaced, including the ampersand (&).

Assume the following variable assignment statement has been executed by the NCL procedure before displaying the sample panel:

```
&VARIABLE = Data
```

&VARIABLE is the only variable within a field which contains the words ‘other data’. Where both field justification and variable alignment are used, the padding character applies to both, as shown by the last line of the example for field character !. The sample panel is displayed as follows:

```
Data      other data
     Data other data
  Data    other data
Data.....  other data
.....Data other data
..Data...  other data
.........................Data.....  other data
```

Input Padding and Justification

Fields to which the PAD and JUST operands of the #FLD statement are applied can be defined as input fields. If an input field is primed with data during the display process, the alignment of data within that field when displayed is as described in the previous section on *Output Padding and Justification*, except that JUST=CENTER is treated as JUST=LEFT. When Panel Services processes input from the screen, input fields defined using the PAD and JUST operands are specially processed using the following rules:

- Trailing blanks and pad characters are stripped off, unless the pad character is numeric.
- If JUST=RIGHT is specified for the field, then leading blanks and pads are stripped off (including numeric pads).
- If JUST=ASIS is specified for the field, then trailing blanks and pads are stripped off, but leading blanks and pads remain intact.

```
#NOTE      This sample panel definition gives examples of the
#NOTE      use of input padding and justification.
#FLD # TYPE=INPUT
#FLD $ TYPE=INPUT JUST=RIGHT
#FLD @ TYPE=INPUT PAD=< JUST=LEFT
#FLD ? TYPE=INPUT PAD=> JUST=RIGHT
#FLD / TYPE=INPUT PAD=0 JUST=LEFT
#FLD } TYPE=INPUT PAD=1 JUST=RIGHT
#VAR01           +
$VAR02           +
@VAR03           +
?VAR04           +
/VAR05           +
}VAR06           +
```

Assume the following variable assignment statements are executed by the NCL procedure before displaying the sample panel:

```
&VAR01 = Walt
&VAR02 = Tom
&VAR03 = Dick
&VAR04 = Harry
&VAR05 = John
&VAR06 = Vicky
```

The sample panel is displayed as:

```
WALT
                                              TOM
DICK<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>HARRY
JOHN0000000000000000000000000000000000000000
1111111111111111111111111111111111111VICKY
```

If control is passed back to the NCL procedure without any data entered into the input fields, the variables are set to the following values:

```
&VAR01 = WALT
&VAR02 = TOM
&VAR03 = DICK
&VAR04 = HARRY
&VAR05 = JOHN
&VAR06 = VICKY
```

Note

If the line John000... had been modified, it would have been padded to the right with zeros.

The variable values are translated to uppercase because the default for input fields is CAPS=YES.

Processing with Light Pens/Cursor Select

Panel Services lets you use both light pens and the cursor select key. Such fields are termed *selector pen detectable*, or SPD fields.

Specify an SPD field by the TYPE=SPD operand on the #FLD statement. An SPD field can be regarded as an input field and the processing and formatting options apply accordingly. However, you cannot enter data into an SPD field—it is protected.

An SPD field must nominate a single variable (minus the &). This field can contain data to be displayed when the panel is displayed. It is set to the word SELECTED if you do either of the following:

- Press the light pen against the screen anywhere in the field.
- Press the CURSOR SELECT key when the cursor is positioned anywhere in the field.

In accordance with hardware requirements, a field specified as TYPE=SPD must start with either ampersand (&), question mark (?), or a blank ( ). These characters are termed *designator characters* and have the following meaning:

- A question mark (?) designates a selection field. If you choose this field, the ? is changed to a greater-than symbol (>) by the hardware to show successful selection (it can be de-selected by re-selecting it once more).
- An ampersand (&) designates the first format for an attention field—select this field in the normal way. (Selecting this field is also equivalent to pressing the ENTER key, which transmits all modified screen data.)
- A blank designates the second format of an attention field. It operates like the & field, but modified data is not transmitted.

The designator character can be followed by one or more blanks and then the name of the variable (without the &) that is to receive notification of the selection. For example:

```
#NOTE This sample panel definition gives an example of the
#NOTE use of SPD fields.

#FLD / TYPE=SPD
/? SELECTION1
```

If you use a light pen or the cursor select key to select this field, the variable &SELECTION1 is set to the value SELECTED on return to the NCL procedure.

Mixing SPD Fields with Normal Input Fields

While you can mix TYPE=SPD fields with TYPE=INPUT fields, some care must be taken.

A normal input field (TYPE=INPUT) lets you key data into it. This data is transmitted to the system when the ENTER key or a Function key is pressed. If an attention SPD field (designated by the blank selection character) is used to replace the ENTER key, data entered into normal input fields is not transmitted, and the variables associated with those fields are set to null. Therefore, it is recommended that when you mix SPD fields with normal input fields, you use only SPD fields specifying ? or & designator characters.

Hardware Restrictions

When using a light pen, you must observe some hardware requirements if you define multiple fields on the one line:

- A minimum of three blank or null characters must precede an SPD field.
- A minimum of three blanks or nulls must follow displayed data in the field before the next field starts.

Note

These restrictions do not apply if you are using the CURSOR SELECT key.

Intercepting Function Keys

By default, Panel Services intercepts certain function key actions in an identical manner to the rest of the system, as follows:

F2 or F14	Screen split operation
F3 or F15	Terminate NCL procedure
F4 or F16	Terminate NCL procedure
F9 or F21	Screen split/swap operation

While the use of these keys is transparent to NCL procedures invoking full-screen panels, this might not always be desirable. A procedure might want to intercept all function keys and invoke alternative functions.

Panel Services offers two levels of function key interception which can be requested using an &CONTROL verb, before issuing the &PANEL statement where they are to apply. To simplify processing, you can perform optional function key mapping, where F13 to F24 are mapped to their F1 to F12 counterparts.

&CONTROL PFKSTD

PFKSTD stops the interception of F3/15 and F4/16, which normally terminate the invoking NCL procedure. Using these keys returns you to the NCL procedure with the appropriate value set in the &INKEY system variable. F2/F14 and F9/F21 operate as normal, providing screen split/swap facilities.

&CONTROL PFKALL

PFKALL lets you allocate alternative functions to all Function keys, or to block screen splitting, if required. PFKALL stops all Function key interceptions and returns to the NCL procedure with the appropriate value set within &INKEY. In this case, screen split and swap facilities are not available unless your NCL procedure issues the appropriate SPLIT or SWAP command when the associated keys are pressed.

&CONTROL NOPFK

Returns PFKSTD or PFKALL to standard operation.

&CONTROL PFKMAP

This option can simplify the number of keys which a procedure must allow for.

PFKMAP maps F13 to F24 against their counterparts before presenting them to the NCL procedure in the &INKEY system variable. When this option is in effect, the procedure does not have to cater for F13 to F24—F13 is presented as F1, F14 is presented as F2, and so on.

&CONTROL NOPFKMAP

NOPFKMAP turns off function key mapping.

Function keys are presented to the procedure unchanged, so that all function keys are available for separate uses.

Note

Terminals under EASINET control always operate as though &CONTROL PFKALL is in effect, and cannot operate in split screen mode.

Using Panels on Different Screen Sizes

The maximum number of display lines that can be defined for a panel is 62. This does not include control statements. It is possible that a panel may exceed the size of the terminal or the size of the current operational window (if operating in split-screen mode). If this happens, Panel Services truncates the panel.

An invoking NCL procedure can use the &LUROWS and &LUCOLS system variables to determine the dimensions of the processing window. The &ZROWS and &ZCOLS system variables can be used to determine the dimensions of the physical terminal. The &ZCURSFLD and &ZCURSPOS system variables are used to determine the actual field which contains the cursor. Using these variables, a procedure can tailor its processing accordingly.

The &LUROWS Variable

The &LUROWS system variable is provided to let an NCL procedure test the number of screen lines currently available for displaying a full-screen panel.

When in split screen operation, this is the number of lines available within the current screen window. Use &LUROWS for multi-screen output displays to determine the maximum number of lines that can be displayed in the available operational window.

You are responsible for subtracting any fixed overheads associated with displaying the panel, such as heading lines, and so on, and for operating within the available space. You should also allow for small windows, where there might not be sufficient lines to display data.

The &LUCOLS Variable

The &LUCOLS system variable lets an NCL procedure test the number of screen columns currently available for displaying a full-screen panel. &LUCOLS can be used for multi-screen output displays to determine the maximum width that can be displayed in the available operational window.

You should cater for small windows, where there might not be sufficient columns to display data.

The &CURSCOL Variable

The &CURSCOL system variable is set on returning from an &PANEL statement and is used to determine the column where the cursor was positioned when the last entry was made.

If operating in split screen mode, the value in &CURSCOL is relative to the start of the operational window regardless of where that window is positioned on your screen. If the last entry was caused by the INWAIT timer expiring, the value returned in &CURSCOL is unreliable.

The &CURSROW Variable

The &CURSROW system variable is set on returning from an &PANEL statement and can be used to determine the row where the cursor was positioned when the last entry was made.

If operating in split screen mode, the value in &CURSROW is relative to the start of the operational window regardless of where that window is positioned on your screen. If the last entry was caused by the INWAIT timer expiring, the value returned in &CURSROW is unreliable.

Determining the Field Location of the Cursor

The &ZCURSFLD and &ZCURSPOS system variables are used to determine the actual field location of the cursor. This is useful for providing context sensitive help.

The &ZCURSFLD system variable contains the name of the field the cursor was in. This applies to TYPE=INPUT and TYPE=OUTVAR fields only—output fields have no name.

The &ZCURSPOS system variable provides the offset within that field where the cursor was positioned.

Controlling Cursor Positioning

The cursor position on a panel is controlled either implicitly by Panel Services (for example, to identify a field in error), or by an explicit request from the procedure issuing the panel.

For explicit cursor positioning, use the #OPT statement CURSOR operand. Either specify the name of an input field defined on the panel, or supply screen co-ordinates as a row and column number.

- If using the name of an input field, enter the name of the variable minus the ampersand, as defined in the panel to receive any data entered in that field. Define the CURSOR operand as:

  ```
  #OPT CURSOR=&CURSOR
  ```

  Then assign the required field name into the &CURSOR variable before displaying the panel. For example:

  ```
  &CURSOR = FIELD2          -* note omission of ampersand
  &PANEL MYPANEL
  ```

- Alternatively, specify explicit cursor positioning by supplying the row and column where the cursor is to be positioned. The CURSOR operand of the #OPT statement can also be used, but then the row and column co-ordinates must be specified. For example:

  ```
  #OPT CURSOR=&ROW,&COL
  ```

  Row and column co-ordinates are then set from the procedure before displaying the panel. For example:

  ```
  &ROW = 10
  &COL = 30
  &PANEL MYPANEL
  ```

  Note

  If the row and column co-ordinates that you specify lie outside the boundaries of the current operating window, the cursor is positioned on row 1, column 1 of the window.

A procedure can also influence the implicit positioning of the cursor by Panel Services. Use the &ASSIGN SETERR operand to let the procedure identify one or more fields which can be classified as being in error.

Alternatively, a procedure can use the #OPT panel statement ERRFLD operand to identify a particular field which is in error.

Cursor Positioning Hierarchy

Cursor location is determined in the following sequence:

1. The first field in error detected by internal validation
2. A field identified as being in error by #OPT ERRFLD
3. The first field designated as being in error by &ASSIGN OPT=SETERR; any field designated by &ASSIGN OPT=SETERR which is also identified by #OPT CURSOR takes precedence
4. A field or position identified by #OPT CURSOR=*loc*
5. The first input field processed top to bottom, left to right
6. The upper left-hand corner of the panel (row 1, column 1 of the window)
7. If the selected field cannot be displayed within the current window dimensions, the cursor is positioned at the upper left-hand corner of the panel (row 1, column 1 of the window)

Dynamically Altering Panel Designs (PREPARSE)

When a panel is designed, the location and layout of input and output fields within it is normally fixed when the panel is created using Edit Services. This means the field characters that define the location of fields (by default %, +, or _), are positioned as required and remain fixed. It is the data or variables within those fields that then change.

However, under some circumstances, you might need to dynamically alter the attributes of fields, or to add or delete entire fields. For example, you might want the color and highlighting for a field to change depending on the data content. (This might be necessary if you are designing a panel that monitors network status, where you want to vary the color of a field to alert an operator).

Some scope for doing this exists by supplying variable data for use on #FLD statements where the variable data is used to alter the characteristics of the field. However, the actual addition or deletion of fields cannot be achieved using this technique and it becomes cumbersome if the attributes of many fields must be altered.

Panel Services lets you dynamically create panel definitions by using a *preparsing* concept. Preparsing is requested by the #OPT panel statement PREPARSE operand, and makes a preliminary scan of the required panel lines before building the panel.

During this preliminary phase, field characters are ignored and substitution used to change a panel line in any way you require. Only after preparsing is complete, is the normal panel building process performed.

Substitution normally uses the occurrence of ampersand (&) to indicate the start of a variable string. These strings can be resolved to reflect the content of the variable, as set by the procedure before the &PANEL statement.

The PREPARSE operand has the following format:

```
#OPT PREPARSE=($,S)

#OPT PREPARSE=(!,D)
```

The first character in the parentheses defines the alternative substitution character (other than an &) to be used during the PREPARSE operation.

The second character specifies the alignment option for the display fields affected by PREPARSE substitution. These options (for *Dynamic* and *Static* preparsing) are explained in the following sections.

The alternative substitution character (for example, a dollar sign) is used as the substitution character for the preparse stage of processing. The panel line is scanned for occurrences of the preparse character and the variables are isolated. The variables are then resolved using the values of the corresponding variables set from the procedure. Using an alternative substitution character allows panels to contain a mix of conventional and preparse fields.

Before displaying this panel (see Figure 6-5) the procedure tests the values in the variables &FIELD2 and &FIELD3 and appends the appropriate field character to display the field. In this example, using the > character displays the field in red and reverse video, but using the ? character displays the field in yellow without any other highlighting.

Figure 6-5. Panel Before PREPARSE Substitution

```
#OPT PREPARSE=($,d) INWAIT=60
#FLD > COLOR=RED HLIGHT=REVERSE TYPE=OUTPUT
#FLD ? COLOR=YELLOW TYPE=OUTPUT
%--------------------  Network Monitor  --------------------------
+$FIELD1

+NETWORK STATUS AS AT &TIME ON &DATE3
+NCP1 is currently ............$FIELD2

+NCP2 is currently ............$FIELD3
```

In this example, FIELD2 and FIELD3 are output fields. FIELD1 is an output field too, by default, but the logic below also converts the FIELD1 position to an input command line if the user ID happens to be USER01. The value assigned to FIELD1 actually contains the new field characters that are to be substituted into the panel definition. The preparse function then builds the new fields before the panel is displayed, so that an input field appears on the terminal.

The following logic shows this procedure:

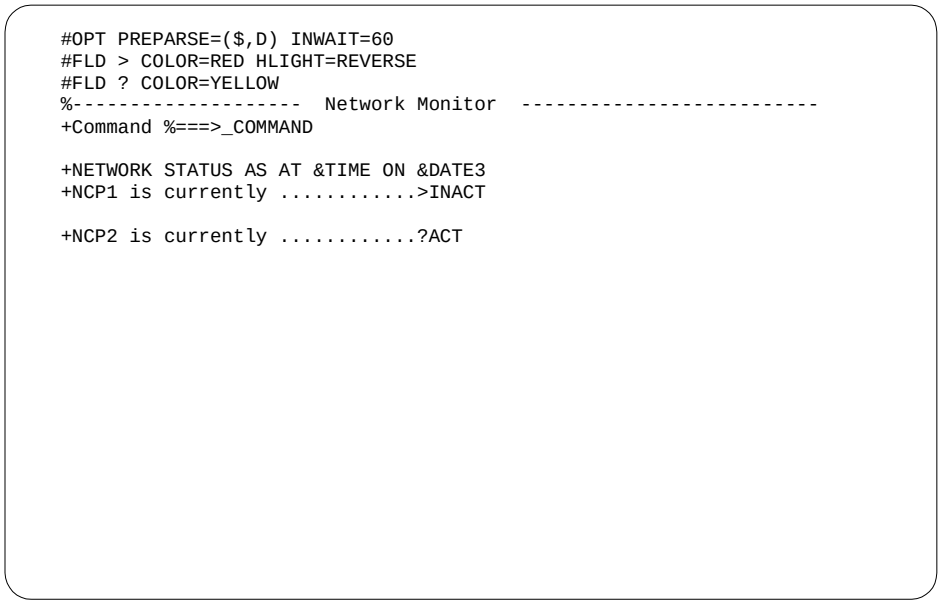
```
&IF &STATUS2 EQ INACT &THEN &FIELD2 = >&STATUS2
&ELSE &FIELD2 = ?&STATUS2
&IF &STATUS3 EQ INACT &THEN &FIELD3 = >&STATUS3
&ELSE &FIELD3 = ?&STATUS3
&IF &USERID = USER01 &THEN &FIELD1 = &STR Command +
                                     %===>_COMMAND
&PANEL MYPANEL
```

Note

Although the variables on the panel start with the preparse substitution character ($), they are still referred to in the NCL procedure as starting with an ampersand.

Preparsing is regarded as a completely separate substitution phase. Therefore, if a preparse character other than an & is designated, this process can substitute data that includes other variables, which are resolved when the standard substitution process for each field is performed.

Figure 6-6. Panel After PREPARSE Substitution



The Dynamic PREPARSE Option

In panel lines with more than one preparse character or field character, the effect of substituting variable data within the line might cause the final text of the line to be longer or shorter than the original text. In this case, fields that follow the substituted data move to the left or right of their original column.

Shifting preparse or field characters to accommodate differing substituted data lengths is the system default, and is called *dynamic preparsing.*

The Static PREPARSE Option

If you want to display a panel where column alignment is important for the presentation, but you are using preparse to substitute data into each line, you could find the columns are not aligned properly in the final display. This occurs because the substituted data varies in length from line to line.

To correct this, use the *static preparse* option. This lets you specify on the #OPT statement that the location of preparse and field characters is to be preserved, despite differing data substitution lengths. Here, data is truncated to fit if the substituted data exceeds the space available to the left of the next preparse or field character.

For example, if a panel uses $ as its preparse character and contains the line:

```
$VAR1  $VAR2
```

An NCL procedure sets the variable as follows:

```
&VAR1 = &STR !ABCDEFGHIJKLMNOPQRSTUVWXYZ
&VAR2 = &STR !12345
```

Dynamic preparse displays:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ 12345
```

Static preparse displays:

```
ABCDE 12345
```

Considerations When Using PREPARSE

Preparsing substitutes by using the specified character. Once the preparse is complete, standard panel field processing proceeds.

Take care that the data substituted does not contain unwanted field characters which introduce unexpected fields if not removed. In Figure 6-5, for example, preparsing may generate an entire line of data that is expected to be output only.

If this data contains any underscore characters (indicating an input field) errors will probably occur because the field format is incorrect. Overcome this by ensuring that the data substituted by preparsing does not contain such unwanted field characters. Varying the default field characters can help here.

Allowing the panel to perform substitution *after* preparse prevents this problem. For example, change the NCL code as follows:

```
&IF &STATUS2 EQ INACT &THEN +
    &FIELD2 = &CONCAT > &STATUS2
```

Displaying Function Key Prompts

The SAA Common User Access (CUA) standards require that a list of function keys and their functions be displayed at the bottom of a panel. The #TRAILER control statement can be used by NCL procedures to nominate lines which appear at the bottom of the panel. The function key prompts are then always displayed at the bottom of the panel regardless of the screen size.

Controlling the Formatting of Input Fields

When a panel is displayed, a data stream is created to format the physical screen as defined in the panel definition. For example, a panel may be displayed repeatedly as a monitor screen or similar, where the display time is controlled by the INWAIT operand. In this example, the INWAIT timer can expire while data or a command is still being entered in the panel. The entered data is ignored and the input field cleared when the panel is refreshed. The cursor position might also change.

You can use the #OPT FMTINPUT statement to avoid this problem by letting the procedure determine when input fields are formatted. So, if you are entering data when the screen is refreshed, and the FMTINPUT=NO specification is used, the entered data remains and you can complete entry unaffected.

By varying the FMTINPUT operand setting (through a variable set from within the procedure) from YES to NO, the procedure can *toggle* input field formatting on and off. For example:

```
#OPT FMTINPUT=&FMT
```

The first time a panel is displayed, it should always specify FMTINPUT=YES to ensure that the physical screen is correctly formatted. On subsequent refreshes (usually after INWAIT expires), FMTINPUT=NO can be used.

For example:

```
.FMTYES
   &FMT = YES
   &GOTO .DISPLAY
.FMTNO
   &FMT = NO
.DISPLAY
   &PANEL MYPANEL
   &IF .&INKEY EQ .  &GOTO .FMTNO
   .
   .
   .  standard processing
   .
   .
   &GOTO .FMTYES
```

Note

The FMTINPUT operand is designed to work in conjunction with the INWAIT facility and must be used with care, or panel errors can result.

Allowing Long Field Names in Short Fields

Panel Services panel definition screens look similar to the panel which is to be displayed. The panel definition screen contains attribute characters followed by the field data (for TYPE=OUTPUT fields) or variable names (for TYPE=INPUT or TYPE=OUTVAR fields). The next field attribute character is placed at the start position of the next field. Unfortunately this means that input fields cannot be any shorter than the variable name which is to contain the input data.

To overcome this, you can use the #ALIAS control statement to define a short alias name for a variable. The alias name can be used where the variable would have been used. You can define a range or list of variables and refer to them by the same alias name in the panel definition.

Retrieving Panels from Panel Libraries

Panels are created using an online editor and are stored in a panel library. When a user is defined, the panel library, or sequence of panel libraries that they are to use, is defined. This is called the *panel library path*. When required, panel specifications are retrieved from a library in the user's current path.

To eliminate overheads associated with retrieving the panel from the library, an in-storage queue of active panels is maintained. When a panel is first referenced, it is retrieved from a panel library and stored on the active panel queue.

Thereafter, the panel is retrieved from the active panel queue without reference to the panels library. If one of these panels is modified (using the online editor), the old copy is removed from the active panel queue so that the next reference retrieves the updated panel from the panel library path.

Note

If a panels library is being shared by more than one SOLVE system, the old version of a modified panel is only removed from the active panel queue of the system on which the panel change has been made. The other systems continue to use the old panel until it is rolled off the active panel queue by other panels being used in the system. The LIBRARY REFRESH command can be used to drop all panels loaded from a library from the active panel queue.

The SHOW PANELS command can be used to list the panels that are retained in the active panel queue.

You can tailor the number of panels that can be retained on the active panel queue by using the SYSPARMS MAXPANEL operand. This is described in the SYSPARMS appendix in the *Management Services Implementation and Administration Guide*.

Displaying Panels on OCS Windows

Full-screen displays can be invoked from an OCS window. OCS windows normally operate in roll mode from top to bottom of the window.

Any NCL process executing in the NCL processing environment associated with an OCS window can issue &PANEL in an attempt to take over the window and place it in full-screen mode.

OCS always grants this bid for the display area if it is operating in its usual rolling mode. It places the window in full-screen mode and initiates queueing of any messages that are directed to the OCS window while it is operating in full-screen mode.

On completion of the NCL procedure that has taken over the window, or when the procedure issues an &PANELEND statement, the OCS window reverts to standard roll mode operation and any queued messages are displayed.

If the number of queued messages exceeds the size of the OCS window, the window is automatically placed in AUTOHOLD mode to ensure you have time to observe all messages.

NCL Processes Competing Against OCS for the OCS Window

If an NCL process executing in an OCS window's NCL processing environment issues an &PANEL statement while the OCS window is in its usual roll mode, OCS (the current owner of the window) always allows the panel to be displayed.

If an NCL process attempts to issue an &PANEL statement when the OCS window is in Holding or Autohold On mode, the &PANEL statement is suspended. The OCS window is placed into FS-HOLD mode and requires operator input before the NCL process panel is allowed to take over the window. At this stage, (Holding or Autohold) the OCS window is logically considered to be in an &CONTROL NOSHAREW condition. That is, OCS owns the presentation area and is not prepared to release it.

Operator input releases the Holding or Autohold mode; OCS immediately switches to the logical &CONTROL SHAREW condition and allows the panel to be displayed.

Competition Between NCL Processes for an NCL Environment Window

Many NCL processes can execute concurrently in an NCL processing environment. If the NCL environment is associated with a window, any process can issue &PANEL in an attempt to take over the window's presentation area to show you their particular panels.

As indicated above, at all times there is a logical owner for the presentation area represented by the window. For an OCS window this is OCS itself most of the time, but when an NCL process issues &PANEL it acquires logical ownership of the window and OCS operates in the background.

The same applies for other NCL processing environments. For example, the Primary Menu is initially owned by the Primary menu NCL process. If that process STARTs other NCL processes which then execute within the same NCL processing environment, these other NCL processes can *bid* for the window to display their own panels by issuing an &PANEL.

Once a process succeeds in becoming the (temporary) owner of a window, it can indicate its willingness to give up its ownership through the &CONTROL SHAREW option.

If the current owner of the window is executing with &CONTROL SHAREW in effect, it will automatically give up its ownership to any other process that issues &PANEL. The previous owner is then logically stacked behind the new owner. When the new owner ends or issues &PANELEND the previous owner's panel is redisplayed.

One of the most common usages of asynchronous NCL processes competing for a window is in an OCS environment. A user can have several independent NCL processes executing in the NCL processing environment associated with an OCS window. These processes might monitor the status of various items in the network and report to the operator when an error condition occurs, by issuing an &PANEL statement to takeover the window.

These processes are written to use &CONTROL NOSHAREW, preventing any other process from stealing the window from them once they displayed their panel. On acknowledgement of their panel by the operator the process issues &PANELEND to release their ownership. The window then reverts to the next waiting process or to OCS.

Note

If a panel is defined with INWAIT=0, indicating that the issuing procedure regains control immediately the panel is displayed, the system guarantees to display the panel before the issuing procedure is allowed to continue. This might mean waiting for some other process to release control of the window.

Using Asynchronous Panels

NCL allows you to write procedures that interact with a display area either *synchronously* or *asynchronously.* If you operate in synchronous mode, when a screen panel is issued the procedure is suspended until input is received from the device or until a timeout (as specified by INWAIT) expires.

This mode of operation suits applications where direct interaction with an operator is expected. The INWAIT function also allows interval-based dynamic updating of screen displays.

Synchronous mode operation does not suit applications which require dynamic updating of screen displays as events occur, or circumstances in which screen displays might have to be updated without necessarily involving any operator input.

To solve these requirements there is a mode of panel operation called *asynchronous* operation.

Asynchronous Operation Concepts

Asynchronous panel operation lets you write an NCL procedure that can issue a panel for display but then continue execution without being suspended to wait for operator input or timeout expiry.

This means that you can develop procedures to drive dynamically updated event panels to display new event information as it happens (rather than at fixed intervals) while retaining the ability to interact with an operator. It also lets you update operator information screens at any time without requiring operator input to trigger resumption of processing.

Invoking an Asynchronous &PANEL Operation

To issue a panel in asynchronous mode, code the TYPE=ASYNC operand on the &PANEL statement. Your procedure is suspended until the nominated panel is scheduled for display, and then control returns from the procedure to the statement immediately following the &PANEL statement.

At this stage, the system variable &RETCODE is set to indicate the result of the &PANEL statement.

The values in &RETCODE are described below and are equivalent to the return codes set if you have specified &CONTROL PANELRC. See the section, *Monitoring Panel Return Codes,* on page 6-11 for further details.

&RETCODE can contain one of the following values:

0 The display operation has not been performed because the nominated panel has already been displayed and input is now available from that earlier display.

This happens if your procedure repeatedly updates a panel but, between updates, the operator enters some input into a field on the panel and causes an input operation by pressing ENTER or a function key. The input variables defined for the panel now contain the updated values as entered by the operator.

4 As for &RETCODE = 0, except no input fields have been changed, so the operator has only pressed ENTER or a function key. Again, the display operation does not take place.

12 The display operation has been scheduled. The panel is displayed either for the first time or to update an earlier display of the same panel.

The meaning of &RETCODE = 12 is completely different from the same return code if a synchronous &PANEL statement is issued.

Waiting for Input

Long-running NCL procedures do not execute continuously. At some point in their logic they go into a wait state, waiting for some event to occur that triggers them to start processing again. A typical example of such an event is input received from the operator.

With synchronous &PANEL operation, the &PANEL statement implies a wait-for-input condition. With asynchronous &PANEL operation, the panel statement itself does not wait for input from the terminal, so another mechanism is needed which tells the procedure when input has been received. The mechanism used is &INTREAD and the procedure's *dependent request queue*. For information about the dependent request queue, see Chapter 2, *NCL Concepts*.

When an asynchronous panel is displayed and something happens on the window (for example, the operator enters input to the panel), a special message is delivered to the procedure's dependent request queue in the format:

`N00101 NOTIFY: PANEL EVENT: event-type RESOURCE:panelname`

where *event-type* describes the type of input activity that is being notified.

When the procedure reaches a point in its logic where it needs to wait for input from the terminal, it issues an &INTREAD TYPE=REQ statement. This automatically places it in a wait state until a *request message* arrives on its dependent request queue.

When input is entered by the operator, the N00101 message is delivered to the dependent request queue, satisfies the &INTREAD statement and the procedure wakes up. By examining the contents of the N00101 message the procedure can determine that it is a notification of a panel event occurring on the asynchronous panel.

It is important to note that the message is only a *notification* that a panel event has occurred. The action taken by the NCL procedure after reading the message depends on the event. The possible events and associated actions are:

INPUT	Input has been received from the terminal. For the input to be made available to the NCL procedure in the associated panel input variables, it is necessary for the procedure to issue an &PANEL TYPE=ASYNC statement.
CHANGE	The terminal window conditions have been changed. This could indicate that a redefinition of the window dimensions has occurred (a SPLIT or SWAP operation has taken place).
BCAST	A broadcast has been sent to the panel. For the broadcast to appear on the asynchronous panel currently displayed, it is necessary for the procedure to issue another &PANEL statement.
TAKEOVER	Another NCL process has attempted to take over the window. The NCL process is only notified of this event if it is the window owner and &CONTROL NOSHAREW is in effect. To allow another NCL process to acquire ownership of the window, the NCL procedure can issue a &PANELEND or a &CONTROL SHAREW statement.

If an INPUT, CHANGE, or BCAST event occurs, the NCL procedure very often issues an &PANEL statement to obtain the input to the asynchronous panel. This statement must specify the name of the panel being displayed when the &INTREAD statement was issued. The &PANEL statement completes with the appropriate &RETCODE return code value as described above. If an &PANEL statement specifying another panel name is issued after the N00101 message is read, the input is no longer available.

Coordinating Other Processing with Input Notification

The fact that input from a panel can be notified not by completion of the &PANEL statement but instead by the arrival of a trigger message on the procedure's internal request queue means that a procedure has the ability to wait for more than one source of input to act as a trigger for it to resume processing.

For example, consider a procedure that has the role of maintaining a dynamically-updated screen display which has to be refreshed with new information as soon as it arrives, but which can receive an input command from the screen.

Such a procedure needs to be able to listen for new information that is to be displayed as well as listening for input received from the screen display.

By designing the procedure so that it receives all its input via &INTREAD TYPE=ANY, it can receive new display messages from its *dependent response* queue (for example from other procedures operating in its dependent environment) and notification of screen input via its *dependent request queue*.

This concept therefore allows a procedure to have a single point in its logic at which it can wait for multiple different forms of input, and so synchronize its processing of different input streams.

Controlling Input Field Initialization

When displaying an asynchronous panel, it is possible to receive output directed to the panel while input is being entered. This could result in partially updated panel input fields being reset when the panel is redisplayed to reflect the received output.

As described in the section detailing the panel control statements later in this chapter, the FMTINPUT operand of the #OPT statement in a panel definition determines if input fields are formatted when a panel is displayed. This operand can be used to control input field re-initialization for both synchronous and asynchronous panels.

In the case of asynchronous panels, if the FMTINPUT operand is not specified in the panel definition, the input fields are reset only the first time the panel is displayed or when the panel is redisplayed after the operator has signalled the end of input (by pressing ENTER or a defined Function key). This means that input can be entered continuously by the operator while the panel output fields are being rewritten as output is directed to the terminal. If the FMTINPUT operand is specified in the panel definition, input field initialization is processed as defined by the operands to #OPT statement.

Managing I/O Contention

Using the CDELAY operand of the &PANEL statement, it is possible to prevent output to a terminal in input mode being delayed, irrespective of the value of the SYSPARMS CDELAY operand. This is advisable in situations where output can affect the structure of a panel being displayed.

For example, members might be asynchronously inserted into a selection list while selections are being made by the operator. In this situation, the chosen selection variable may be associated with a certain entry before the list is updated and a associated with a different entry afterward. Specifying CDELAY=N synchronizes the arrival of the output with its appearance on the panel.

For a more detailed description of the CDELAY concept, see the section about the SYSPARMS CDELAY operand in your *Management Services Implementation and Administration Guide*.

Panel Control Statements

Optional control statements can precede a panel to specify the particular requirements for that panel. These are:

#ALIAS	Defines alternative field names
#ERR	Defines the action to be taken for an error condition
#FLD	Defines or modifies a field character's attributes
#NOTE	Provides installation documentation (this is ignored during processing)
#OPT	Defines optional operational requirements
#TRAILER	Defines trailing lines for the panel

Control statements included within panel definitions must precede the displayable portion of the panel which is determined by the first line that is not a control statement. Control statements must start in column 1 of the lines on which they appear.

Note

You cannot include comments on the same line as a control statement (except #NOTE).

#ALIAS

Function:

Defines an alternative name for input variables.

```
#ALIAS        name
             {  VARS=prefix*[ RANGE=(start,end) ]  |
                VARS={ vname | (vname,vname, ...vname) } }
```

Use:

To allow the panel definition to contain alternative names for variable names in TYPE=INPUT and TYPE=OUTVAR fields.

This facility is useful if you want to have short fields with long variable names. Each reference to *name* in the panel definition is regarded as a reference to the next name from the list of VARS specified.

Operands:

name

Specifies the alias name that will appear in the panel definition. Whenever this name occurs in a field declared as TYPE=INPUT or TYPE=OUTVAR on the #FLD statement, panel services logically replaces it with the next available name from the vars list.

The name can be from one to eight characters in length. The first character must be an alphabetic or national character. The remaining characters, if any, must be alphanumeric or national characters.

The same name can be used on multiple #ALIAS statements. The variable names are simply added to the end of the list of names to which the alias name refers.

VARS=*prefix [RANGE=(*start,end*)] |**
VARS=(*vname,vname, ..., vname*)

Specifies the list of names to replace the alias name in the panel definition. Each time the alias name is encountered in the panel definition, it is replaced by the next available name from this list. The formats of the operands associated with VARS= are as follows:

VARS=*prefix** denotes that the variable names used will be *prefix*1, *prefix*2, and so on. The RANGE= operand can be specified to indicate a starting and ending suffix number. The default is RANGE=(1,64). The format *prefix** cannot be used in conjunction with other variable names on the same #ALIAS statement.

VARS=*vname* is the name of a variable, excluding the ampersand (&).

Examples:

```
#ALIAS Z VARS=LONGNAME
#ALIAS Z123 VARS=(SATURDAY,SUNDAY)
#ALIAS AVAR VARS=LINE* RANGE=(10,20)
```

Notes:

Multiple #ALIAS statements can be used for the same alias name if insufficient space is available on a single statement**.**

If an alias name appears in the panel definition after all the variable names in the alias list have been used up, the alias name itself appears in the panel.

Symbolic variables can be included in the #ALIAS statement. Variable substitution is performed prior to processing the statement, using variables available to the NCL procedure at the time the &PANEL statement is issued.

See Also:

The #FLD panel control statements.

#ERR

Function:

Defines the action to be taken during error processing.

```
#ERR    [  INTENS={ HIGH | LOW  } ]
        [  { COLOR | COLOUR } = { BLUE | RED | PINK | GREEN |
                        TURQUOISE | YELLOW | WHITE | DEFAULT }  ]
        [  { HLIGHT | HLITE } = { USCORE | REVERSE | BLINK | NONE }  ]
        [  ALARM={ YES | NO  } ]
```

Use:

The #ERR statement is a panel control statement that determines the processing required when a panel is being redisplayed following an error condition.

An error condition can be detected either by Panel Services internal validation or by the processing NCL procedure. If detected by internal validation (and &CONTROL PANELRC is not in effect), error processing is automatically invoked by Panel Services. If detected by the processing NCL procedure, error processing is invoked in one of two ways:

- By using the &ASSIGN OPT=SETERR verb
- By nominating the name of the variable that identifies the invalid input field on the ERRFLD operand of the #OPT statement. This is dynamically invoked by using a symbolic variable with the ERRFLD operand and setting this variable to the name of the variable (minus the ampersand) that identifies the field in error.

See the section, *Handling Errors*, on page 6-14, for a detailed discussion on using this technique.

When #ERR processing is initiated, the cursor is positioned to the first field in error and the panel is redisplayed, applying the attributes defined on the #ERR statement to the fields in error. This technique provides the panel user with a simple means of drawing the terminal operator's attention to the field in error. This is particularly effective on color terminals where the color of any field in error can be altered for the duration of the error and reverts to normal when the error condition is rectified.

Normally only one #ERR statement is defined. However, if required to accommodate the operands, multiple statements can be defined. They can be defined in any order. However, as with #OPT, #FLD, and #NOTE statements, any #ERR statement must precede the start of the panel, which is determined by the first line that is not a control statement.

Operands:

INTENS={ HIGH | LOW }

Determines the intensity of the error field when displayed. The INTENS operand is ignored for terminals with extended color and highlighting when either the COLOR or HLIGHT operand is specified.

HIGH specifies that the field is displayed in double intensity.

LOW specifies that the field is displayed in low or standard intensity.

{ COLOR | COLOUR } = { BLUE | RED | PINK | GREEN | TURQUOISE | YELLOW | WHITE| DEFAULT }

Determines the color of the field. It applies only to IBM terminals with seven-color support and Fujitsu terminals with three- or seven-color support.

The COLOR operand is ignored if the terminal does not support extended color. This enables COLOR to be specified on panels that are displayed on both color and non-color terminals. COLOR can be used in conjunction with the HLIGHT operand.

For Fujitsu terminals that support extended color datastreams, but support only three colors, the following color relationships are used:

Specified	**Result** (on Fujitsu three-color terminal)
GREEN	GREEN
RED	RED
PINK	RED
BLUE	GREEN
TURQUOISE	GREEN
YELLOW	WHITE
WHITE	WHITE
DEFAULT	GREEN

Fujitsu seven-color terminals are treated the same as IBM seven-color terminals.

The DEFAULT keyword indicates that the color of the field is to be determined from the INTENS operand. This is particularly useful if you want to set the color from an NCL procedure (that is, COLOR=&COLOR is specified and the NCL procedure can set the &COLOR variable to DEFAULT).

{ HLIGHT | HLITE } = { USCORE | REVERSE | BLINK | NONE }

Applies only to terminals with extended highlighting support, and determines the highlighting to be used for the field.

The HLIGHT operand is ignored if the terminal does not support extended highlighting, so HLIGHT can be specified on panels that are displayed on terminals that do not support extended highlighting. HLIGHT can be used with the COLOR operand.

When NONE is specified, the HLIGHT operand is ignored and no extended highlighting is performed for this field.

ALARM={ YES | NO }

Determines if the terminal alarm is to be rung when the panel is displayed with an error condition. This works independently of the ALARM operand on the #OPT control statement.

Examples:

```
#ERR COLOR=RED HLIGHT=REVERSE ALARM=YES
#ERR COLOR=YELLOW HLIGHT=BLINK INTENS=HIGH
```

Notes:

Only those attributes defined on the #ERR statement are modified for the field in error. All other attributes associated with the original field, such as internal validation, remain intact.

Symbolic variables can be included in a #ERR statement. Variable substitution is performed before processing the statement, by using variables available to the NCL procedure at the time the &PANEL statement is issued.

When &CONTROL PANELRC is in effect, internal validation does not automatically reshow the panel with the error message, and so on. In this case, the procedure regains control following the &PANEL statement with the &RETCODE system variable set to 8 to indicate that an error has occurred. The &SYSMSG variable contains the text of the error message that describes the error and the &SYSFLD variable contains the name of the field in error. This name is the name of the variable in an input field that would receive the data entered into that field. See the section, *Internal Validation*, on page 6-17, for more information on this type of processing.

The &ASSIGN statement SETERR option provides a mechanism for assigning #ERR field attributes to multiple (input field) variables before displaying a panel. This allows you to accept input from a number of different fields on a panel, validate all the fields and then redisplay the panel with all incorrect fields displayed with the #ERR attributes. This shows the user all the errors at one time, rather than field by field.

See Also:

The #OPT, #FLD, and #NOTE panel control statements and the &ASSIGN statement in the *Network Control Language Reference*.

#FLD

Function:

Defines or modifies a panel definition field character.

```
#FLD    {  c | X'xx' }
        [  BLANKS={ TRAIL | NONE | ANY } ]
        [  CAPS={ YES | NO } ]
        [  { COLOR | COLOUR } = {  BLUE | RED | PINK | GREEN |
                  TURQUOISE | YELLOW | WHITE | DEFAULT } ]
        [  CSET={ ALT | DEFAULT } ]
        [  EDIT={ ALPHA | ALPHANUM | DATEn | DSN | HEX |
                  NAME | NAME* | NUM | REAL | SIGNNUM | TIMEn }  ]
        [  { HLIGHT | HLITE } = { USCORE | REVERSE | BLINK | NONE }  ]
        [  INTENS={ HIGH | LOW | NON } ]
        [  JUST={ LEFT | RIGHT | ASIS | CENTER | CENTRE } ]
        [  MODE={ SBCS | MIXED } ]
        [  NCLKEYWD={ YES | NO } ]
        [  OUTLINE={  {L R T B} | BOX } ]
        [  PAD={ NULL | BLANK | char } ]
        [  PSKIP={ NO | PMENU } ]
        [  RANGE=(min,max) ]
        [  REQUIRED={ YES | NO } ]
        [  SKIP={ YES | NO } ]
        [  SUB={ YES | NO } ]
        [  TYPE={ OUTPUT | INPUT | OUTVAR | SPD | NULL } ]
        [  VALIGN={ NO | LEFT | RIGHT | CENTER | CENTRE } ]
```

Use:

The #FLD statement is a panel control statement used to tailor the operational characteristics of a panel.

When a panel is defined, it is made up of a number of lines, which in turn are made up of a number of fields. Each field commences with a field character, which appears as a blank on the panel when displayed. Each field character determines the attributes that are to be associated with the field following the field character itself. A field is delimited by the next field character or the end of the panel line. Fields cannot wrap from one line to the next.

The first field on a line always starts in column 1. If no field character is defined in the first position of the line, the attributes of the second of the three standard field characters (usually a plus sign (+), TYPE=OUTPUT, INTENS=LOW) are forced, and they replace any non-field character incorrectly placed in this position.

Before parsing, the #FLD statement is scanned and variable substitution is performed. This makes it possible to dynamically tailor any of the options or operands on the statement.

You can specify as many #FLD statements as required, and you can define them in any order. However, as with #OPT, #ERR and #NOTE statements, all #FLD statements must precede the start of the panel, which is determined by the first line that is not a control statement.

Operands:

***c* | X'*xx*'**

The field character:

c is the character that is used in the panel definition to identify the start of the field. This is known as a *field character.* This must be a single non-alpha and non-numeric character. Any special character (for example, an exclamation mark) can be used, except an ampersand (&), which is reserved for use with variables.

X'*xx*' is the *hexadecimal* value of the field character. Use this notation to specify any value in the range X'01' to X'3F'. Do not use values which correspond to alphanumeric characters , or X'0E' (shift in) or X'0F'(shift out).

Although the panel editor does not allow you to enter non-displayable hex attributes (X'01' to X'3F') in the body of the panel, you can use the preparse option to prime the field character value in the panel before issuing the &PANEL statement.

The first #FLD statement to reference a particular field character defines a new character. Subsequent statements referencing that same field character modify or extend the attributes of the field character. Three standard field characters (%, +, _, unless altered by the #OPT statement) are provided. If a default field character (usually % + or _) is referenced, it is the same as extending or modifying the attributes of an existing field character.

If no additional operands are defined following a new field character then the following defaults apply:

```
TYPE=OUTPUT INTENS=LOW
```

No special attributes or internal validation apply.

BLANKS={ TRAIL | NONE | <u>ANY</u> }

For input fields this determines the format the entered data must take. By default, a field can contain imbedded blanks (BLANKS=ANY). Specification of this operand ensures that the entered data does not contain imbedded blanks and contains only trailing blanks (TRAIL) or no blanks at all (NONE). This operand works independently of the REQUIRED operand. For optional fields this operand can still be specified to ensure that any data entered is in the correct format. If &CONTROL PANELRC is not in effect, BLANKS=TRAIL is specified, and the data is in error, Panel Services redisplays the panel with the &SYSMSG variable set to:

```
INVALID IMBEDDED BLANKS
```

If BLANKS=NONE is specified and the data is in error, Panel Services redisplays the panel with the &SYSMSG variable set to:

```
INCOMPLETE FIELD
```

If &CONTROL PANELRC is in effect, control is returned to the NCL procedure for error handling instead of being handled totally by Panel Services. In this case, &SYSFLD contains the name of the field in error and &SYSMSG the error message text. See the section, *Internal Validation*, on page 6-17, for further information.

CAPS={ <u>YES</u> | NO }

Applies to input fields only and determines if entered data is to be converted to upper case before passing it to the NCL procedure in the nominated variable. Conversion to upper case is also performed for the data associated with an input variable before displaying the panel. This does not affect the current contents of the variable, unless the data is modified and entered by the operator. Output fields are displayed exactly as defined and are not subject to upper case conversion.

Uppercasing for CAPS=YES fields uses the language code of the user region to select the character set that is used as the basis of the translation. Where the language code of the *user* is not one of the supported values, the language code of the system is used. Where the language code of the *system* is not one of the supported values, a translation based on EBCDIC codes is performed.

Note

The effect of CAPS=NO can be negated if the variable that receives the data is used in an assignment statement (for example, &A = &DATA) within the processing NCL procedure, as data can be converted to upper case before performing the assignment; see the &CONTROL UCASE option.

The CAPS operand is ignored when operating in a system executing with SYSPARMS DBCS=YES.

{ COLOR | COLOUR } = { BLUE | RED | PINK | GREEN |
TURQUOISE | YELLOW | WHITE | DEFAULT }

Applies only to IBM terminals with seven-color support and Fujitsu terminals with three- or seven-color support, and determines the color of the field.

The COLOR operand is ignored if the terminal does not support extended color. This enables COLOR to be specified on panels that are displayed on both color and non-color terminals. COLOR can be used in conjunction with the HLIGHT operand.

For Fujitsu terminals that support extended color datastreams where only three colors are available, the following color relationships are used:

Specified	**Result** (on Fujitsu three-color terminals)
GREEN	GREEN
RED	RED
PINK	RED
BLUE	GREEN
TURQUOISE	GREEN
YELLOW	WHITE
WHITE	WHITE
DEFAULT	GREEN

Fujitsu seven-color terminals are treated the same as IBM seven-color terminals.

The DEFAULT keyword indicates that the color of the field is to be determined from the INTENS operand. This is particularly useful if you want to set the color from an NCL procedure (that is, COLOR=&COLOR is specified and the NCL procedure can set the &COLOR variable to DEFAULT).

CSET={ ALT | <u>DEFAULT</u> }

Applies to output fields only. The operand determines which terminal character set to use to display the field. If you specify CSET=ALT (or ALTERNATE), you can draw box shapes using the following characters:

e is displayed as

s is displayed as

D is displayed as 

E is displayed as 

M is displayed as 

N is displayed as 

F is displayed as 

G is displayed as 

O is displayed as 

P is displayed as 

L is displayed as 

Note

CSET=ALT supersedes CSET=ASM in version 3.1

EDIT={ ALPHA | ALPHANUM | DATE*n* | DSN | HEX | NAME | NAME* | NUM | REAL | SIGNNUM | TIME*n* }

For input fields this determines additional internal editing to be performed by Panel Services. By default no editing is performed. Specification of this operand ensures that the entered data conforms to the nominated type. If a field is mandatory, then REQ=YES should also be specified.

ALPHA

Accept A to Z only.

ALPHANUM

Accept A to Z, 0 to 9, #, @, and $ only.

DATE*n*

Field must be a valid date. The DATE*n* keyword must correspond to one of the &DATE*n* system variables, and the input field must contain date in the format associated with that system variable. For example, EDIT=DATE5 indicates that the input field must always contain a date in the format corresponding to the &DATE5 system variable (MM/DD/YY).

DSN

Field must be a valid OS/VS format dataset name. If required, a partitioned dataset (PDS) member name or Generation Data Group (GDG) number can be specified in brackets as part of the name.

HEX

Accept 0 to 9 and A to F only.

NAME

Field must commence with alpha (A to Z, #, @, or $) and be followed by alphanumerics (A to Z, 0 to 9, #, @, or $).

NAME*

Field must commence with alpha (A to Z, #, @, or $) and be followed by alphanumerics (A to Z, 0 to 9, #, @, or $) but can be terminated with a single asterisk (*). This allows a value to be entered that can be interpreted as a generic request by the receiving procedure.

NUM

Accept 0 to 9 only.

REAL

Input in this field must conform to the syntax for integers, signed numbers or real numbers, including scientific notation. See the chapter Chapter 5, *Arithmetic in NCL*, for information on real number support.

SIGNNUM

Field must be numeric but can have a leading sign (+ or -).

TIME*n*

Field must be a valid time. The TIME*n* keyword must correspond to one of the &ZTIME*n* system variables, and the input field must be in the format associated with that system variable.

When invalid data is detected and &CONTROL PANELRC is not in effect, standard error processing is invoked by Panel Services and control is not returned to the NCL procedure until the error is corrected.

For EDIT=NUM, the panel is redisplayed with the &SYSMSG variable set to:

```
FIELD NOT NUMERIC
```

For EDIT=REAL, the panel is redisplayed with the message

```
FIELD NOT REAL NUMBER
```

For EDIT=ALPHA, ALPHANUM, HEX, or NAME, the panel is redisplayed with the &SYSMSG variable set to:

```
INVALID VALUE
```

For EDIT=DATE*n*, the panel is redisplayed with the &SYSMSG variable set to:

```
INVALID DATE
```

For EDIT=DSN, the panel is redisplayed with the &SYSMSG variable set to:

```
INVALID DATASET NAME  or INVALID MEMBER NAME
```

For EDIT=TIME*n*, the panel is redisplayed with the &SYSMSG variable set to:

```
INVALID TIME
```

In all cases the terminal alarm is rung and the cursor is positioned to the field in error. If a #ERR statement has been included in the panel definition, processing of the error condition is performed as defined on that statement.

If &CONTROL PANELRC is in effect, control is returned to the NCL procedure for error handling instead of being handled totally by Panel Services. In this case, &SYSFLD contains the name of the field in error and &SYSMSG the error message text. See the section on *Internal Validation* in this chapter for further information.

Note

Use of the EDIT operand might also require the use of the BLANKS operand to ensure that entered data does not include imbedded blanks. Regardless, editing is performed only for the length of the data entered and not for the length of the input field. If the entire field is to be entered, the BLANKS=NONE operand should be specified.

{ HLIGHT | HLITE } = { USCORE | REVERSE | BLINK | <u>NONE</u> }

Applies only to terminals with extended highlighting support, and determines the highlighting to be used for the field.

The HLIGHT operand is ignored if the terminal does not support extended highlighting. This enables HLIGHT to be specified on panels that are displayed on terminals that do not support extended highlighting. HLIGHT can be used in conjunction with the COLOR operand.

The NONE keyword is provided as a no-impact value that can be used when the highlighting of a field is being dynamically determined from the NCL procedure and set using variable substitution of the #FLD statement. When NONE is specified, the HLIGHT operand is ignored.

INTENS={ HIGH | <u>LOW</u> | NON }

Determines the intensity of the field when displayed.

HIGH

The field is displayed in double intensity. High intensity is normally associated with input fields and other important data and its use minimized to maintain its effectiveness.

LOW

The field is displayed in low or standard intensity.

NON

The field is displayed in zero intensity, that is, any data within the field is not visible to the operator.

This is normally used for input fields where sensitive data such as passwords are to be entered. Use of this attribute for output fields is meaningless. Color or extended highlighting attributes are ignored when used in conjunction with this attribute.

JUST={ <u>LEFT</u> | RIGHT | ASIS | CENTER | CENTRE }

For output fields, this determines the alignment of the data within the field after trailing blanks have been stripped. Justification is applied at a field level and should not be confused with VALIGN which applies to the individual variable only:

- JUST=LEFT results in padding to the right
- JUST=RIGHT results in padding to the left
- JUST=ASIS is treated as JUST=LEFT for output fields
- JUST=CENTER results in padding to both the left and the right.

For input fields, justification occurs both when the data is being displayed and when the data is being processed on subsequent entry. When an input field is formatted for display (the value currently assigned to the variable defined in the input field is substituted in place of the variable's name), the data is justified to the left and padded to the right for JUST=LEFT or justified to the right and padded to the left for JUST=RIGHT. JUST=CENTER is treated like JUST=LEFT. For JUST=ASIS, data is positioned exactly as defined in the variable and padding to the right is performed.

On subsequent re-entry, trailing blanks and pad characters are stripped, unless the trailing pad character is a numeric, in which case it is not stripped:

- For JUST=RIGHT, leading blanks and pads are also stripped (including numerics). Use of JUST=RIGHT for input fields might inconvenience terminal operators, as it is necessary to cursor across to the commencement of the data in the field
- For JUST=ASIS, trailing blanks and pads are stripped, but leading blanks and pads remain intact.

MODE={ SBCS | MIXED }

Applies to IBM terminals capable of supporting DBCS datastreams. If a panel is sent to such a device, input fields on the panel that use this #FLD character allow the operator to enter DBCS characters if MODE=MIXED is specified.

IBM DBCS terminals do not allow DBCS character entry in input fields that specify MODE=SBCS (single byte character stream).

Note

This operand does not apply to Fujitsu terminals, which allow DBCS character entry at any time.

NCLKEYWD={ YES | NO }

Specifies whether fields that use this FLD character accept input of words that conflict with NCL keywords. The default is YES. If you attempt to enter any NCL reserved keyword NO causes it to be rejected.

OUTLINE={ { L R T B } | BOX }

Specifies the extended highlighting outlining option required for this field. Any combination of L (left), R (right), T (top), or B (bottom) can be coded. The field is outlined at the top or bottom with a horizontal line and at the left or right border with a vertical line, according to the options specified. Alternatively, you can specify the BOX option, which is equivalent to specifying LRTB. This option is terminal-dependent.

PAD={ NULL | BLANK | *char* }

Applies to INPUT, OUTPUT, and SPD fields.

For output fields, PAD works in conjunction with both the JUST and VALIGN operands, one of which must be specified for PAD to take effect. Determines the pad or fill character to be used when the field is displayed.

The variable substitution process substitutes the data currently assigned to any variables within the field being processed. When substitution is complete, any difference between the length of the field defined on the panel and the length after substitution (after stripping trailing blanks) is padded with the specified PAD character.

For input fields, use of the NULL character ensures that the terminal operator can use keyboard insert mode when entering data. Padding is performed either to the left or to the right, as specified in the JUST operand.

char

Specifies a single character that is to be the pad character (for example, PAD=-).

There is no restriction on the character used, including the use of any of the field characters defined on #FLD statements. Care should be taken when using numeric pad characters, as their use impacts the pad character stripping process on subsequent entry.

Note

Using PAD characters with input fields invokes special processing on subsequent input to ensure that unnecessary pad characters are stripped before returning the entered data in the nominated variable. See the *Input Padding and Justification* and *Output Padding and Justification* sections for detailed information.

PSKIP={ NO | PMENU }

Applies to input fields only and determines if panel skip requests are accepted in this field. A panel skip request is entered in an input field in the format =*m.m*, where *m.m* is a menu selection. When this is entered in an appropriate field, a panel skip to the specified menu selection is performed.

NO

The input field is not scanned for panel skip requests.

PMENU

The input field is scanned for panel skip requests and actioned.

RANGE=(*min,max*)

For numeric fields, specifies the range of acceptable values. The range includes all numbers, from the minimum number (*min*) to the maximum number (*max*). Both *min* and *max* must be specified, and *max* must be equal to or greater than *min*. Use of this operand forces EDIT=NUM. If the entered number falls outside the acceptable range and &CONTROL PANELRC is not in effect, Panel Services redisplays the panel, with the &SYSMSG variable set to:

```
NOT WITHIN RANGE
```

If &CONTROL PANELRC is in effect, control is returned to the NCL procedure for error handling instead of being handled totally by Panel Services. In this case, &SYSFLD contains the name of the field in error and &SYSMSG the error message text. See the section on *Internal Validation* in this chapter for further information.

REQUIRED={ YES | NO }

Specifies whether this is a mandatory field that must be entered by the user. If &CONTROL PANELRC is not in effect, Panel Services rejects any entry by the user unless the mandatory field has been entered. If it is not entered, Panel Services redisplays the panel with the &SYSMSG variable set to:

```
REQUIRED FIELD OMITTED
```

The terminal alarm is rung and the cursor is positioned to the omitted field. If a #ERR statement has been included in the panel definition, processing of the error condition is performed as defined by the #ERR statement. Failure to include the &SYSMSG variable on the panel suppresses this error message. See the section on *Internal Validation* for more details. This operand can be abbreviated to REQ=.

If &CONTROL PANELRC is in effect, control is returned to the NCL procedure for error handling instead of being handled totally by Panel Services. In this case, &SYSFLD contains the name of the field in error and &SYSMSG contains the error message text. See the section on *Internal Validation* in this chapter for further information.

SKIP={ YES | NO }

For output fields only, determines if the skip attribute is to be assigned to the field. The result of using this option is that the cursor skips to the next input field if the preceding input field is entered in full and the intervening output field is specified with the SKIP operand.

Note

This operand is NO by default, because field skipping can unexpectedly place the cursor in the wrong screen window when operating in split screen mode.

SUB={ YES | NO }

For output fields only, determines if variable substitution is to be performed. This operand is normally used only for fields where data contains the & character, which normally results in the current value of that variable being substituted or the variable being eliminated if no value is assigned. This operand is ignored for both INPUT and SPD fields.

TYPE={ OUTPUT | INPUT | OUTVAR | SPD | NULL }

Determines if the field is to be processed as an output-only field (OUTPUT and OUTVAR), input field (INPUT), Selector Pen Detectable (SPD) field, or pseudo input field (NULL).

OUTPUT

A protected field is created, which does not allow keyboard entry. This field can contain a mixture of fixed data and variables. Each variable must commence with an ampersand (&). Substitution of variables is performed using the variables available to the invoking NCL procedure at the time the &PANEL statement is issued. Global variables can be referenced in an output field. Alignment and padding are performed according to the rules defined for the field.

INPUT

An unprotected field is created, which allows keyboard entry. This field must contain a single variable name (without the ampersand). This single variable must immediately follow the field character. System variables and global variables cannot be used in an input field. Subsequent data entered into this field is made available to the invoking NCL procedure in this variable on return from the &PANEL statement. Specification of multiple variables or a mixture of variables and fixed data in an input field results in an error.

OUTVAR

The same as TYPE=OUTPUT, except that an & is inserted by Panel Services between the field attribute and the next character. This means that you can follow a TYPE=OUTVAR field character with a variable name without the ampersand. This facility makes it easy to create fields which switch between input and output under NCL control. For example, a panel might contain the statements:

```
#FLD $ TYPE=&INOUT
+ Record Key .....$RKEY +
```

An NCL procedure would then set the variable &INOUT to control if the data in the variable &RKEY is output only, or if it can be modified by an operator:

```
&INOUT = OUTVAR -* the value is output only.
&INOUT = INPUT -* the Operator can modify the
-* field.
```

Note

A similar effect can be achieved using &ASSIGN OPT=SETOUT.

SPD

A protected field is created in selector pen detectable format. This enables the terminal operator to select the field using either a LIGHT PEN or the CURSOR SELECT key. SPD field characters must be immediately followed by one of the three designator characters (?, &, or a blank), which can in turn optionally be followed by one or more blanks. A single variable with no other fixed data must also be defined within the field. This single variable must be defined without the ampersand (&) and cannot be a system or global variable. The SPD support provided by Panel Services is discussed in detail in the section *Processing with Light Pens/Cursor Select* in this chapter. If selected by the user, the variable nominated in the SPD field is set to the value SELECTED on return to the NCL procedure. If not selected, the variable is set to a null value.

NULL

An unprotected field is created, which allows keyboard entry. However, the field need not contain the name of an input variable to receive data entered in the field, as any data entered by the terminal operator in a TYPE=NULL field is ignored. Display data in this field can be in any format. The NULL option is supplied to accommodate four-color terminals where the field attribute byte is used to determine the color in which the field is displayed (seven-color terminals utilize an extended datastream to set the color). The NULL option indicates that Panel Services is to use an unprotected field attribute in conjunction with the INTENS operand value to determine the color of the field.

VALIGN={ NO | LEFT | RIGHT | CENTER | CENTRE }

Applies to output fields only, and is ignored if specified for an input field. Determines the alignment of data for an individual variable only. This should not be confused with the JUST operand, which applies to field alignment after all variable substitution has been completed. The VALIGN operand is designed to facilitate tabular output without the need to specify many individual field characters on the panel.

The substitution process normally substitutes the data assigned to a variable in place of the variable name. No additional blanks are created or removed during this process. Thus, if the data being substituted is shorter than the name of the variable itself (for example, the variable &OUTPUTDATA is currently set to 5678), then data following the variable name is shifted left to occupy the area remaining after the removal of the variable name. This would of course destroy any tabular alignment where the length of the data for each variable differed. The VALIGN option ensures that the data to the right of the variable is not shifted to the left if the data being substituted is shorter than the variable name. The length of the variable name (including the ampersand) is the important factor and determines the number of character positions to be preserved during the substitution process.

However, data is not truncated and, if the data being substituted is longer than the variable name, then the data to the right is moved to accommodate all the substituted data. VALIGN works in conjunction with the pad character specified on the PAD operand. The pad character is used to fill any difference between the data being substituted and the length of the variable name being replaced.

VALIGN=NO
: No alignment or padding is performed.

VALIGN=LEFT
: Data is aligned to the left and padded to the right. An abbreviation of L is acceptable.

VALIGN=RIGHT
: Data is aligned to the right and padded to the left. An abbreviation of R is acceptable.

VALIGN=CENTER
: Data is centered (or one position to the left for an odd number of characters) and padded both to the left and to the right. An abbreviation of C is acceptable.

Examples:

```
#FLD#   TYPE=INPUT REQ=YES EDIT=NUM COLOR=RED RANGE=(1,3)
#FLD#   BLANKS=TRAIL PAD=_
#FLD#   TYPE=OUTPUT COLOR=&COLOR HLIGHT=&HLIGHT
#FLD(   TYPE=INPUT INTENS=HIGH EDIT=DATE4
#FLD@   HLIGHT=BLINK
#FLD/   TYPE=SPD
#FLD%   JUST=R PAD=- -* supplementing default output char
#FLD_   JUST=ASIS -* supplementing default input char #FLD +
VALIGN=RIGHT JUST=CENTER
                    -* null pad assumed
```

Notes:

Multiple #FLD statements can be used for the same field character if insufficient space is available on a single statement.

Symbolic variables can be included in a #FLD statement. Variable substitution is performed prior to processing the statement, using variables available to the NCL procedure at the time the &PANEL statement is issued.

You can alter the default field characters by using the DEFAULT operand of the #OPT control statement.

You can use the #ERR control statement to greatly simplify the redisplay of a panel to indicate a field in error.

The &CONTROL PANELRC operand can be used to specify that the NCL procedure receives control for further processing when internal validation detects an error in data entered by the operator. When this technique is used, the procedure can determine the field in error (from the &SYSFLD variable) and the error message to be issued (from the &SYSMSG variable), and alter its processing accordingly, including altering the text of the error message in the &SYSMSG variable if required.

See Also:

The #OPT, #ERR, and #NOTE panel control statements.

#NOTE

Function:

Allows user comments in a panel definition.

#NOTE	*any text*

Use:

The #NOTE statement provides a means of placing documentation within a panel definition. The #NOTE statement is not processed and is ignored. Multiple #NOTE statements can be specified. However, as with #FLD, #ERR, and #OPT statements, all #NOTE statements must precede the start of the panel, which is determined by the first line that is not a control statement or #NOTE statement.

Operands:

any text
Any free form user text.

Examples:

```
#NOTE This panel is used by the Network Error Log System
#NOTE INWAIT=60 CURSOR=&CURSORFLD
```

Notes:

As shown in the example above, the #NOTE statement can provide a simple means of temporarily nullifying another control statement, allowing for easy reinstatement when required.

See Also:

The #FLD, #ERR, and #OPT panel control statements.

#OPT

Function:

Defines panel processing options.

```
#OPT    [  ALARM={ YES |_NO_ } ]
        [  BCAST={ YES |_NO_ } ]
        [  CURSOR={ varname | row,column } ]
        [  DEFAULT={ hlu  | X'xxxxxx'  } ]
        [  ERRFLD=varname  ]
        [  FMTINPUT={ _YES_ | NO } ]
        [  IPANULL={ _YES_ | NO } ]
        [  INWAIT=ss.th ]
        [  PREPARSE={ (c,S) | (c,D) } ]
        [  UNLOCK={ _YES_ | NO } ]
        [  MAXWIDTH={ YES | _NO_ } ]
```

Use:

The #OPT statement is a panel control statement used to tailor the processing options for a panel.

Before parsing, the #OPT statement is scanned and any variables are substituted. This makes it possible to dynamically tailor any of the operands on the statement.

Multiple #OPT statements can be specified. However, as with #FLD, #ERR, and #NOTE statements, all #OPT statements must precede the start of the panel, which is determined by the first line that is not a control statement.

Operands:

ALARM={ YES | _NO_ }

Determines whether the terminal alarm is to be rung when the panel is displayed. Dynamic control of the alarm can be achieved by changing the value of the ALARM operand using a variable set prior to issuing the &PANEL statement.

If internal validation has detected an error and the panel is being redisplayed to indicate the error, this operand is ignored and the terminal alarm rung. The #ERR statement can be used to alter the processing performed when an error condition is detected.

BCAST={ YES | _NO_ }

Specifies that the panel is to be redisplayed automatically if a broadcast is scheduled. By default, the only panels that are redisplayed automatically are those that contain one or more of the special broadcast variables, &BROLINE*n*. If BCAST=YES is coded, a broadcast causes the panel to be redisplayed even if it does not contain any of the &BROLINE*n* variables.

CURSOR={ *varname* | *row,column* }

Specifies the name of a variable in either an INPUT or SPD field where the cursor is to be positioned. Alternatively, the precise coordinates for the cursor can be defined as *row,column*.

The rules defining how the cursor is positioned are explained in the section, *Cursor Positioning Hierarchy,* on page 6-31.

The value of *varname* should be the variable name *without* the ampersand, just as used in the INPUT or SPD field (for example, CURSOR=FIELD5).

Where coordinates are specified, *row* must be specified in the range 1 to 62 and *column* in the range 1 to 80. The *row* and *column* values are always relative to the start of the current window and therefore remain unchanged when operating in split screen mode. The &CURSROW and &CURSCOL system variables can be used to determine the location of the cursor on input to the system.

Dynamic positioning of the cursor can be achieved by using a variable or variables (including the ampersand) in place of *varname* or *row,column*. The invoking NCL procedure can set the variables to the name of the field to contain the cursor or the coordinates prior to issuing the &PANEL statement.

If internal validation has detected an error and the panel is being redisplayed to indicate the error, the CURSOR operand is ignored and the cursor is positioned to the field in error.

Specifying *varname* with a name other than the name of a variable used in an INPUT or SPD field results in an error. If coordinates are used and they lie outside the dimensions of the window currently displayed, the cursor is positioned in the upper left corner of the window.

DEFAULT={ *hlu* | X'*xxxxxx*' }

Alters the three standard default field characters. If the #OPT statement is omitted or the DEFAULT operand not used, then three standard field characters are provided for use when defining the panel. They are:

%	=	protected, high-intensity
+	=	protected, low-intensity
_	=	unprotected, high-intensity

It is sometimes necessary to select alternative field characters, for example if the underscore character is required within the body of the panel for some reason.

The DEFAULT=*hlu* operand must always specify three characters. The characters chosen must be non-alpha and non-numeric, that is, any special character except ampersand (&), which is reserved for variables. They must not duplicate another field character, except one already defined as a default. The order of the characters is significant, as the attributes of the standard default characters apply in the order described above.

Therefore specification of `DEFAULT=*+/` results in:

*	=	protected, high-intensity
+	=	protected, low-intensity
/	=	unprotected, high-intensity

You can also specify the default field characters in hexadecimal in the format:

`DEFAULT=X'`*`xxxxxx`*`'`

where each *xx* pair represents a hexadecimal number in the range X‘00’ to X‘FF’. All numbers except X‘00” (null), X‘40’ (blank), and X‘50’ (ampersand), are valid. This even allows alphanumeric characters to be used as field characters. For example, if you specify X‘C1’, any occurrence of the letter A in the panel definition is treated as the default character. However, it is advisable to use hexadecimal values that do *not* correspond to alphanumeric characters.

For example, specification of `DEFAULT='010203'` would result in:

X‘01’	=	protected, high-intensity
X‘02’	=	protected, low-intensity
X‘03’	=	unprotected, high-intensity

ERRFLD=*varname*

Specifies the name of a variable in an INPUT field which is in error and for which error processing is to be invoked, as defined on a #ERR statement. Use of this operand without including a #ERR statement within the panel definition results in an error. The ERRFLD operand provides a simple way of informing Panel Services that the field identified by *varname* is in error. Panel Services displays the panel using the options defined on the #ERR statement. The #ERR statement could indicate that the error field is to be displayed in reverse-video, colored red and the terminal alarm rung. Use of the ERRFLD operand is normally accompanied by the assignment of some error text into a variable appearing on the screen to identify the nature of the error.

This operand is normally specified with the name of a variable (including the &) that is set to null unless an error occurs, in which case the NCL procedure sets the variable to the name of the field in error prior to issuing the &PANEL statement to display the panel.

ERRFLD provides the panel designer with a simple means of changing the attributes of a field (such as color and high-lighting) without needing to resort to dynamic substitution of #FLD statements.

Consider the case where an input field &INPUT1 is found to be in error and the #OPT statement has been defined with ERRFLD=&INERROR. The NCL procedure simply assigns the name of the variable used to identify the input field, in this case INPUT1 (minus the &), into &INERROR and then redisplays the panel.

```
&INERROR = INPUT1
&SYSMSG  = &STR THIS FIELD IS WRONG
&PANEL  MYPANEL
```

Note

In this example the text that identifies the nature of the error has been assigned into the variable &SYSMSG which would be defined somewhere on the panel.

The same effect can be achieved by using the &ASSIGN OPT=SETERR verb. This allows more than one field to be marked as in error.

Note

&ASSIGN OPT=SETERR is effective only if &CONTROL FLDCTL is in effect.

FMTINPUT={ YES | NO }

Determines if input fields are to be formatted when a panel is displayed. This is a specialized option that is designed to be used in conjunction with INWAIT. When processing with INWAIT, the time interval could expire at the instant when data is being entered by the operator. If the same panel is redisplayed to update the screen contents, the data entered by the operator is lost as the new panel is written. FMTINPUT can be used to bypass formatting of input fields and hence when the panel is redisplayed only output fields are written. The value of this operand is normally assigned to a variable from within the NCL procedure and changed between YES and NO as required (#OPT FMTINPUT =&YESNO). Care must taken when using this facility, as incorrect use of FMTINPUT=NO can result in validation errors. Ideally, a panel should be displayed initially with FMTINPUT=YES and only when the INWAIT timer expires would it be redisplayed with FMTINPUT=NO.

IPANULL={ YES | NO)

The default, YES, specifies that if the panel is displayed with the INWAIT option, and the time specified on the INWAIT expires so that control is returned to the procedure without any panel input, or input is caused by a PA key, all variables associated with the panel input fields are to be set to null value.

If you do not want input field variables to be erased if INWAIT completes or a PA key is pressed, specify IPANULL=NO.

INWAIT=*ss.th*

Specifies the time in seconds and/or parts of seconds that Panel Services is to wait for input from the terminal prior to returning control to the NCL procedure following the &PANEL statement. By default, the system, having displayed a panel, waits indefinitely for input. This is not always desirable, as is the case where a terminal is performing a monitoring function where input can be infrequent or never occur. If INWAIT is utilized and the specified time elapses, control is returned to the NCL procedure with all input or SPD variables set to null. If input is made during the time interval, the time period is cancelled and standard processing proceeds. See the section, *Controlling How Long a Panel is Displayed*, on page 6-9, for more details.

The maximum value that can be specified for INWAIT is 86400.00 seconds (24 hours).

Specification of part seconds is possible. For example:

```
INWAIT=.5
INWAIT=20.5
INWAIT=.75
```

Any redisplay of a panel, caused for example by use of the clear key, causes the panel to be redisplayed and the time interval reset.

Specification of internal validation options, such as REQUIRED=YES, is ignored if the time interval expires before input is received.

Specification of INWAIT=0 or INWAIT=0.00 indicates that no input is to be accepted, and control is returned to the NCL procedure immediately after the panel has been displayed. In this case the period that the panel remains displayed is determined by subsequent action taken by the procedure.

The invoking NCL procedure can determine, by testing the &INKEY system variable, if the INWAIT time elapsed or if data was entered. &INKEY is normally set to the character value of the key pressed by the operator to enter the data (for example, ENTER or F1). If the INWAIT time interval elapsed and no entry was made, the &INKEY variable is set to null. If processing with &CONTROL PANELRC in effect, &RETCODE is set to 12 to indicate that the INWAIT timer has expired.

Note

INWAIT is ignored for asynchronous panels.

LSM=YES | NO

If you do not want the LSM to control a particular panel, then code #OPT LSM=NO in the panel definition. The entire panel is then written to the terminal each time, but there is a reduction in storage, which can become significant if very large numbers (thousands) of EASINET terminals are being supported.

PREPARSE={ (c,S) | (c,D) }

Preparsing provides a means for dynamically modifying the location of field characters in a panel. Normally, the position of field characters (as defined by the #FLD control statement) is determined when the panel is created by Panel Services and remains fixed until the panel is modified.

Although the attributes of each field character (such as the color of the field) can be modified by the use of variables in the #FLD statement, this technique is limited in the number of variations that can be achieved.

The PREPARSE operand requests that Panel Services performs a preliminary substitution scan of each panel line prior to processing the line for field characters. The PREPARSE operand specifies a substitution character (*c*) that will be used to determine where substitution will take place. This character is processed in exactly the same manner as an ampersand (&) is processed during standard substitution.

The ability to specify a character other than an ampersand means that preparsing does not impact standard substitution when it is performed following preparsing. Preparsing can be used to alter a field character that appears in a particular position, thereby allocating a new set of attributes to the field, or to create entire new fields (or complete lines) that in themselves contain the required field characters. The section, *Dynamically Altering Panel Designs (PREPARSE)*, on page 6-32 gives examples of this technique.

(*c*,S)

Indicates that the character *c* will be used as the preparse character for the panel, but that the Static Preparse Option will apply during preparse processing. This prevents the movement of preparse or field characters during the substitution process. This option is useful when panels are being dynamically modified to hold data which can vary in length but is to be displayed in columns. If necessary, substituted data can be truncated if it is too long to fit into its target field without overwriting the next occurrence of a preparse or field character on the same line.

(*c*,D)

Indicates that the character *c* will be used as the preparse character for the panel, but that the Dynamic Preparse Option will apply during preparse processing. The dynamic option allows the movement of preparse or field characters to the left or right of their original position to accommodate differing lengths of data being substituted into the panel.

UNLOCK={ YES | NO }

Determines if the terminal keyboard will be unlocked when the panel is displayed. Specification of UNLOCK=NO prevents entry of data by the terminal operator and is normally used in conjunction with the INWAIT operand where a panel is being displayed for a short period, prior to progressing to some other function.

MAXWIDTH={ YES | NO }

When MAXWIDTH=NO is specified or defaulted, the panel display is limited to the standard 80-column width. If you are designing a panel to take advantage of a wider screen, for example a model 5 terminal, specify MAXWIDTH=YES to allow the full width of the screen to be used.

Examples:

```
#OPT DEFAULT=#$%
#OPT INWAIT=60 CURSOR=&CURSORFLD
#OPT CURSOR=IN1 ALARM=YES
#OPT ALARM=&ALARM PREPARSE=($,D)
#OPT ERRFLD=&INERROR
#OPT INWAIT=.5 UNLOCK=NO PREPARSE=($,S)
#OPT CURSOR=5,75
#OPT CURSOR=&ROW,&COLUMN FMTINPUT=&FMT
```

Notes:

Multiple #OPT statements can be used if required.

Symbolic variables can be included in a #OPT statement. Variable substitution is then performed prior to processing the statement.

A panel is redisplayed automatically following use of the CLEAR key. Control is not returned to the invoking NCL procedure.

The attributes of the standard default characters can be modified using a #FLD statement that adds additional attributes (such as color) or alters existing attributes.

See Also:

The #FLD, #ERR, and #OPT panel control statements.

The description of Preparse processing in this chapter.

#TRAILER

Function:

Provides a means of placing specified lines at the end of the screen, regardless of screen size.

```
#TRAILER      [ START | END ]
              [ POSITION={ YES | NO } ]
```

Use:

The #TRAILER statement can be used to position function key prompts at the bottom of the screen.

Indicate the start of the trailer lines with a #TRAILER START statement. Then enter the lines to appear at the end of the screen, followed by a #TRAILER END statement.

Operands:

START

Indicates the start of the lines to be placed in the trailer. Each line following this line until a #TRAILER END statement or another control statement such as #FLD is placed in the trailer.

END

Indicates that this is the end of the lines to be placed into the trailer. There must have been a #TRAILER START statement earlier in the panel definition.

No other operands can be specified on a #TRAILER END statement.

POSITION={ YES | NO }

Specifies if the trailer lines are to be displayed. The values available are:

YES

The trailer lines are displayed on the final lines of the physical screen.

NO

This value can be used to suppress the display of the trailer lines, even though they remain in the panel definition.

Examples:

```
#TRAILER START
%This appears on the last line of the panel
#TRAILER END
```

Notes

- The #TRAILER statements must appear before the first panel line in the definition. If you want to preparse the lines, you must place the #TRAILER statements after the #OPT PREPARSE= statement.
- The field attribute characters which you use in the trailer lines can be defined before or after the trailer lines in the panel.
- The trailer lines cover any panel lines that would otherwise have been displayed.
- The trailer lines are positioned so that they end at the bottom of the physical screen if the window starts at the top of the screen.

7

NCL File Processing

This chapter discusses the following topics:

- UBD File Formats
- Working with UDBs
- Working with Files
- Working with Data

UBD File Formats

The term UDB refers to any file processed with the NCL file processing statements. Three types of physical file come under this heading:

- Mapped format files
- Unmapped format files
- Delimited format (or UDB format) files

Mapped Format Files

Mapped format processing can be used on any file which is arranged in a manner that is describable to Mapping Services using a *map*. A map is used to describe a file in terms of structures or components understood by Mapping Services.

Mapped format file access is the preferred method of file processing in NCL. The advantages of mapped format processing are:

- Allows for transparent data to be placed in the file
- Allows for upward compatibility
- Combined with Mapping Services, allows NCL to operate at a logical level, even when dealing with complex data formats

Using the Default Map

A default mapping, using the $NCL map can be used to maintain NCL variable data in a file. Individual data records up to 32K and containing transparent variable data, are supported in this manner. Each variable exists in a vector format and can be isolated using length fields and data tags.

When using the $NCL map, the file statements reference one or more NCL variables in the normal manner. The actual structure of the record can be understood by referring to the distributed $NCL map.

Using Other Maps

When a retrieval is being carried out from a file using a mapped format, the contents of the file records are read into a *mapped data object* (MDO). A map is attached to the MDO to provide NCL with a means of interpreting its contents. The &ASSIGN verb can then be used to reference certain sections of the data by name and extract these from the MDO into NCL tokens (variables). Mapping Services does the work of locating the sections of data which are being extracted by using the map. By doing the work of locating components within the data, Mapping Services can save a considerable number of NCL statements, particularly for files that have a complicated format, and also for files that contain non-printable data.

Because the map definition is a separate entity within Management Services, it is also possible that subsequent changes to the file format will only require changes to be made to the map definition, and not to the NCL code itself.

Mapping Services can be used to define maps capable of interpreting most data formats.

Unmapped Format Files

NCL can also be used to process records from VSAM files for which no map is available.

Unmapped mode processing is generally used in applications where UDBs created or used by NCL procedures are to be processed off-line by other systems.

NCL file processing statements allow the NCL user to indicate that a particular UDB is to be processed in *unmapped mode*, causing NCL to bypass any attempt to identify individual fields within records read from or written to the file.

While still allowing all the simplicity of access to the file, unmapped mode processing means the NCL user must know the structure of records on the file. Consequently more attention must be paid to this aspect of file processing than is the case for the simpler delimited format files.

Printing (Using Unmapped Format File Support)

Unmapped format is for processing data of any format including binary or non-printable data. It is also useful for report generation where the data is to be directed to a system printer. In an MVS system the unmapped format is ideal for generating reports that are to be routed to JES SYSOUT for printing either on local or remote printers. NCL provides operands that help control report formatting (see the &FILE ADD statement).

If required, a mapped format or delimited format file can be processed in unmapped mode.

The processing mode (UDB format, mapped, or unmapped) for any file can be swapped at any time during NCL procedure processing and this might become necessary if special key structures are being used.

Delimited Format Files

Delimited format is the term given to the default format of NCL structured files up to the introduction of the new &FILE verb in V2.2. This format is supported for upward compatibility, but has the restriction that data placed in the file must be displayable.

Note

Mapped format ($NCL) is the default if a file is opened without specifying a format.

Data in UDB format files is stored in records that contain any number of fields to a maximum total record size of 32K, which is the maximum statement size that can be processed by NCL. Each field is isolated from the next by a high-value (X'FF') *field separator* and the last field in a record is followed by a field separator. The length of a field is determined by the length of data supplied by the NCL user. The occurrence of two consecutive field separators indicates the presence of a null field that contained no data when the record was created, even though the field concerned is logically part of the record. The total length of any record includes the key and all field separators.

Figure 7-1. Format for a Record in a UDB Format File

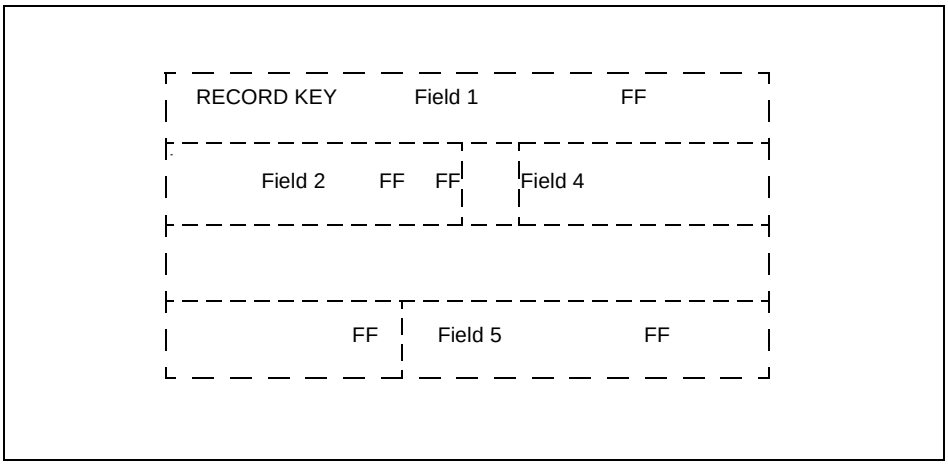


Figure 7-1 shows an example of a UDB format record that contains five fields, field 3 being a null field and represented by the two consecutive field separators at the end of field 2.

This approach eliminates the need to define a file structure and its subsequent modification, should the length of fields change. The rules for field separators within UDB format files are as follows:

- A field separator follows each field
- No field separator follows the key
- A trailing field separator follows the last field, unless it is a key-only file with no data other than the key
- Two consecutive field separators indicate a field of 0 length (a null field)

Multiple File and Alternate Index Support

NCL supports the concurrent processing of multiple files (including a mixture of delimited format, mapped format, and unmapped format files) in addition to the use of VSAM alternate indexes that allow records to be retrieved using keys based on different parts of the data. For example, a UDB used to maintain Help Desk problem information might contain a sequentially allocated problem number, in addition to the name of the resource to which the problem was related. Using alternate indexes, it is possible to retrieve records in both problem number order and resource name order.

Working with UDBs

UDB support is a standard function of NCL. Netmaster Databases (NDBs) provide more advanced facilities for users with requirements for more complex, high-performance data management within their NCL procedures. These data management facilities are described in later chapters of this manual.

NCL supports the following techniques for VSAM *user database* (UDB) files:

- Processing by specific key (KSDS)
- Processing by partial or generic key (KSDS)
- Sequential processing (KSDS), in ascending or descending key order
- Retrieval for update or deletion (KSDS)
- Deletion by specific key (KSDS)
- Multiple record deletion by generic key (KSDS)
- Sequential processing, forwards or backwards (ESDS)
- Emptying (resetting) of datasets (KSDS)
- Processing of SYSOUT datasets (MVS only)

NCL uses the VSAM access method for its speed and flexibility. The high-level NCL implementation shields the user from all VSAM complexities.

UDBs are keyed VSAM datasets (KSDS) or sequential VSAM datasets (ESDS). For keyed datasets, any key length from 1 to 255 characters can be selected. Standard VSAM restrictions regarding duplicate keys apply.

Assigning UDBs to Management Services (OS/VS)

In OS/VS systems a DD card can be included in the SOLVE system JCL for each UDB that is to be used by any NCL procedure. Any valid DD name can be chosen. If the DD name of a UDB starts with the letters *UDB*, the UDB is opened automatically during the initialization of Management Services.

DD cards with DD names that do not start with UDB are not opened automatically and must be identified as UDB definitions through use of the UDBCTL command. UDBCTL commands can be included in the system initialization procedures NMINIT and NMREADY or can be entered from an OCS window, and should be coded as follows:

```
UDBCTL OPEN=ddname options
```

where *ddname* is the DD name of the DD card defining the UDB and options are any special processing attributes to be applied to this UDB.

If special options are to be applied when opening a UDB, it is recommended that their DD names do not start with the characters UDB, otherwise UDBCTL commands must be first used to close the UDB before re-opening them with their required options.

Alternatively, the UDBDEFER initialization parameter can be used to delay the opening of system UDBs.

Dynamic Allocation of UDBs

In MVS and VM/SP, systems UDBs can be allocated dynamically using the ALLOCATE command described in your *Management Services Command Reference*. UDBs allocated in this manner must be identified to Management Services as UDBs through use of the UDBCTL command as described above. Dynamically allocated UDBs can be deallocated (using the DEALLOCATE command) after the UDBCTL command has been used to close them.

Dynamic allocation of UDBs offers greater flexibility than the fixed assignment of files through the system's execution JCL and allows files to be brought on-line after the system has been initialized, thus offering the potential for sharing those files with other systems.

Assigning UDBs to Management Services (VSE)

Assignment of UDBs to be used in VSE systems requires a DLBL card to be included in the execution JCL for each UDB that is to be processed.

The file name used on the DLBL statement is chosen by the installation. No automatic initialization of UDBs is performed in VSE.

For each DLBL statement included in the execution JCL for a UDB, a UDBCTL command must be executed identifying the UDB to Management Services. These UDBCTL statements should be included in the NMINIT or NMREADY initialization procedures or entered from an OCS window and should be coded as follows:

```
UDBCTL OPEN=filename options
```

where *filename* is the file name of the DLBL statement defining the UDB and *options* are any special processing attributes to be applied to this UDB.

Preparing to Use a UDB

To create and use a UDB, do this:

Step 1. Determine the format of the dataset, the length of the records it is to contain and the size of the key to be used. If records are to be added or updated on-line, the key must commence in the first position of the record (unless the dataset is to be processed in unmapped mode). The amount of space required should also be determined at this stage. Remember to take the presence of field separators into account when deciding on record length. Normally field separators have little effect on overall record length, but they can become significant when a record contains numerous very short fields.

Step 2. Use the IDCAMS VSAM utility to define the dataset. The DEFVSAM member in the distribution library contains an example of defining a VSAM dataset.

Step 3. Add a DD or DLBL statement to the Management Services JCL, or allocate the file dynamically with the ALLOCATE command (MVS and VM/SP only).

If the UDB that is to be used is an unmapped format file that already exists and contains data, only the last three steps need to be taken.

Step 4. If allocated using the ALLOCATE command, use the UDBCTL command to OPEN the UDB and assign any special processing attributes that might be required.

Step 5. Make the file available for processing by assigning a logical file ID with the UDBCTL command.

Step 6. Include the appropriate &FILE OPEN statement in the NCL procedure.

After these steps have been performed, the file is available for processing using standard NCL statements.

UDB Initialization

Management Services attempts to open UDBs when the system UDB is initialized or when a UDBCTL OPEN command is processed. If the open fails, the system internally uses IDCAMS to verify the dataset and then retry the open. Therefore it is not necessary to include verification steps in the system JCL.

Initialization of KSDS UDBs

If the dataset requires an initial load, the system loads a single record with a key of all X‘00’ and then deletes it. If this load fails, the dataset is closed and is classified as unusable, and is blocked from further processing until the problem has been rectified. Use the SHOW UDB command to determine the cause of the error.

Initialization of ESDS UDBs

Empty ESDSs identified as UDBs are initialized by loading a single record which has the format:

```
N28510 VSAM INITIAL LOAD PERFORMED AT hh.mm.ss ON
day-dd.mon-year
```

This record is always the first record of an ESDS UDB, unless the ESDS already contains records when opened by Management Services. NCL procedures that reference the UDB should ignore this first record. If preparing for unmapped processing of an NCL-created ESDS UDB, use IDCAMS to REPRO the dataset skipping, the first record.

Writing to SYSOUT as an ESDS

An ESDS UDB that is actually a SYSOUT dataset (OS/VS systems only), whether defined in the Management Services execution JCL or allocated dynamically using the ALLOCATE command, does not require initialization and therefore this processing is bypassed. SYSOUT datasets can therefore be written directly from NCL procedures without an initialization record appearing on the output.

Note

Management Services does not load alternate indexes. You must do this using the IDCAMS BLDINDEX function.

In OS/VS systems, Management Services automatically opens any UDBs identified by DD cards in the execution JCL that start with the letters UDB. UDBCTL OPEN commands are needed only if UDBs are dynamically allocated or are defined by DD cards in the JCL which start with a string other than UDB. Automatic opening can be suppressed by the UDBDEFER initialization parameter.

Controlling UDB Performance and Resource Usage

The techniques used by NCL should ensure efficient processing of VSAM files. Additional performance gains can be obtained by the allocation of additional buffers and processing strings. This is achieved using the JCL AMP statement subparameters on the DD or DLBL statement for the file or options on the UDBCTL command:

BUFNI	the number of index buffers to be allocated by VSAM
BUFND	the number of data buffers to be allocated by VSAM
STRNO	the maximum number of concurrent strings VSAM is to use

By default, Management Services allocates 2 data buffers, 3 index buffers and 2 processing strings unless alternative values are provided as described above.

This buffer allocation applies per string—that is, allocation of a complete set of buffers is performed by VSAM for each concurrent position held on the UDB. Where the usage of a UDB is such that a large number of concurrent accesses to the UDB might be possible, care must be taken that VSAM buffer allocations do not lead to storage shortages affecting the performance of other Management Services functions. Also, in cases such as these, NCL procedures should be written to avoid the maintenance of generic UDB processing environments over long periods.

While varying these parameters can offer significant performance benefits, they should only be changed if the impact on VSAM processing is clearly understood. Incorrect changes can impose severe storage overheads which could impact the operation of other system components.

Adding Records to a UDB

The &FILE ADD statement is used to add new records to a UDB. The &FILE PUT statement can also be used to add records. However, whereas &FILE ADD receives an error indication if a record with a like key exists, &FILE PUT will replace a record with a like key. If an NCL procedure is known to be creating new records then &FILE ADD should be used.

Before a record can be added, the key of the record must be identified. This is done using the KEY= or the KEYVAR= operand on the &FILE statement. The KEY= operand can be used in conjunction with the ADD and PUT operands.

The &FILE ADD statement is used to supply the data of the record to be added to the UDB. Depending on the format of the UDB this could be a text string, tokens, or an MDO.

The success of the &FILE ADD statement can be tested using the &FILERC (file return code) system variable. If the &FILE ADD statement was not successful, the &VSAMFDBK system variable will contain a standard VSAM completion code indicating the type of error that occurred. For example:

```
&FILE OPEN ID=MYFILE FORMAT=DELIMITED
&FILE ADD ID=MYFILE KEY='RECORD1' VARS=A* RANGE=(1,7)
&IF &FILERC NE 0 &THEN &WRITE ERROR CODE=&VSAMFDBK
```

SYSOUT Considerations

A SYSOUT dataset (OS/VS) can be defined as a UDB and be the subject of &FILE ADD (or &FILE PUT) statements. The SYSOUT dataset can be defined via a DD card in the execution JCL or it can be allocated dynamically using the ALLOC command. In either case it appears to Management Services as a VSAM ESDS.

NCL procedures can use &FILE ADD or &FILE PUT statements to write to SYSOUT UDBs. The UDB should be treated as a mapped or unmapped format file and processed using the unmapped format option.

NCL supports both ANSI and machine print control characters. However, it is recommended that you use ANSI characters because they are easier to use.

In MVS systems, NCL determines the record format (RECFM) of a SYSOUT dataset to decide if the dataset is to be supported with ANSI print control characters (RECFM=A) or machine control characters (RECFM=M). When the SYSOUT dataset is dynamically allocated the PRTCNTL operand can be used to specify which format is required.

Formatting SYSOUT Output

When processing SYSOUT datasets in ANSI format the &FILE PUT/ADD statement supports specification of formatting options such as skipping to new pages, line spacing, bolding, and underscoring using the PRTCNTL operand. As such the print character is never formatted by the user as part of the output data. When processing with machine control characters, the user is responsible for formatting the output data with the appropriate control character as the first character of the output data.

When processing an ANSI format SYSOUT file (RECFM=A or PRTCNTL=A) additional sub-operands of PRTCNTL allow left or right justification or centering of the data. In such cases the logical width of the report is determined by the logical record length (LRECL) of the dataset. This too can be specified on the ALLOC command using the LRECL operand. The width specified does not include the print control character which will be allowed for and added internally.

If the dataset is the MVS internal reader (INTRDR, which must be allocated dynamically) it can be treated as an unmapped format UDB, in which case written records are interpreted as JCL. In this way, jobs can be submitted for execution.

When submitting jobs in this way, the job number of the submitted JCL stream can be obtained by issuing &FILE GET ID=*name* END. The job number is returned in the variable &ZJOBNUM.

Updating Records in a UDB

The &FILE PUT statement is also used to update records in a UDB. A record can either be updated by first reading it with an &FILE GET statement and then replacing it with an &FILE PUT statement or by replacing it directly with an &FILE PUT statement. The method chosen will be determined by the application. Regardless of the method to be utilized the key of the record is identified using the KEY= or KEYVAR= operand on the &FILE statement.

If the key was specified on the &FILE GET statement, there is no need to specify it again on the &FILE PUT statement because the retrieved key remains current for the file.

If the possibility exists that multiple updates for the same record can be attempted concurrently, precautions must be taken to ensure that updates are not lost or inadvertently overwritten.

The success of the &FILE PUT statement can be tested using the &FILERC system variable. If the &FILE PUT statement is not successful the &VSAMFDBK system variable contains a standard VSAM completion code indicating the type of error that occurred. For example:

```
&FILE OPEN ID=MYFILE FORMAT=DELIMITED
&FILE GET ID=MYFILE KEY='RECORD1' VARS=B*
&IF &FILERC NE 0 &GOTO .NORECORD
    .
    .
    .  update data
    .
    .
&FILE PUT ID=MYFILE ARGS RANGE=(1,7)
&IF &FILERC NE 0 &WRITE DATA=ERROR CODE=&VSAMFDBK
```

&FILE PUT will replace a record which already exists. Where multiple concurrent updates are possible the NCL procedure must ensure that it has exclusive control of a record prior to issuing the &FILE PUT statement to update it. This can be achieved by first reading the record with an &FILE GET statement that specifies the OPT=UPD operand. This operand indicates that exclusive control of the record is required. If the record is already in use, the &FILE GET statement fails with &FILERC set to 8 and &VSAMFDBK set to 14. The NCL procedure can then take alternative action, such as delaying processing for a short period before retrying the &FILE GET. Having obtained the record the &FILE PUT statement can be issued to complete the update.

Note

When using &FILE GET with the UPD option, the NCL procedure should complete processing of the record as quickly as possible. It is not good practice to obtain exclusive control of a record and then issue a full-screen panel which waits for operator input. This can delay other users for excessive periods.

An example of a procedure using the &FILE GET ... OPT=UPD operand follows. This example issues a one second delay if exclusive control of the record cannot be obtained, and then retries the &FILE GET.

```
&FILE OPEN ID=MYFILE FORMAT=DELIMITED
.RETRY
   &FILE GET ID=MYFILE KEY='RECORD1' OPT=UPD ARGS
   &IF &FILERC EQ 0 &THEN &GOTO .UPDATE
   &IF &FILERC EQ 8 AND &VSAMFDBK EQ 14 &THEN &GOTO .WAIT
   &ENDAFTER &WRITE DATA=ERROR CODE=&VSAMFDBK

.WAIT
   &DELAY 1
   &GOTO .RETRY
.UPDATE
   .
   .
   .  update data
   .
   .
&FILE PUT ID=MYFILE ARGS RANGE=(1,7)
&IF &FILERC NE 0 &THEN &WRITE DATA=ERROR CODE=&VSAMFDBK
```

Deleting Records from a UDB

The &FILE DEL statement is used to delete records from a UDB. Two techniques can be used:

- Deletion by specific key—the full key of the record to be deleted is supplied. Only the record with this key is deleted.
- Deletion by generic key—a partial key is supplied. Any record that matches this partial key (KEQALL) or is equal to or greater than this partial key (KGEALL) is deleted.

The KEY= or KEYVAR= operand can be used to specify the full or partial key to be used for the delete.

Deletion of a single record can be achieved as shown in the following example:

```
&FILE OPEN ID=MYFILE FORMAT=DELIMITED
&FILE DEL ID=MYFILE KEY='RECORD1'
```

If a partial key is provided for a single record delete, NCL deletes the first record on the file that matches the partial key.

If a group of records is to be deleted, a partial key that identifies the first record in the group must be specified on the KEY= or KEYVAR= operand. Then either of the following two values must be set for the OPT= operand.

KEQALL	Delete this record and all following records that have keys equal to the partial key provided.
KGEALL	Delete this record and all following records that have keys equal to or greater than the partial key provided.

The success of the &FILE DEL statement can be tested using the &FILERC system variable. If the &FILE DEL statement was not successful, the &VSAMFDBK system variable will contain a standard VSAM completion code indicating the type of error that occurred. For example:

```
&FILE OPEN ID=MYFILE FORMAT=DELIMITED
&FILE DEL ID=MYFILE KEY='RECORD' OPT=KEQALL
&IF &FILERC NE 0 &WRITE DATA=ERROR CODE=&VSAMFDBK
```

The number of records deleted by an &FILE DEL statement is returned in the system variable &FILERCNT, which can then be used, for example, as feedback information for display on a panel. That is:

```
&FILE OPEN ID=MYFILE FORMAT=DELIMITED
&FILE DEL ID=MYFILE KEY='RECORD' OPT=KGEALL
&WRITE &FILERCNT RECORDS DELETED
```

Note

&FILERCNT remains until the next &FILE DEL, or the file is closed, or processing changes to a different file. In the last case, &FILERCNT changes to reflect the number of records deleted on the file that is now being processed.

Use of generic delete functions is of value when large numbers of records are to be deleted from a UDB and gives better performance than the use of multiple single record deletes.

Retrieving Records from a UDB

The &FILE GET statement is used to retrieve records from a UDB. A number of techniques can be used:

- Retrieval by specific key—the full key of the required record is supplied. Only a record with this key will be returned.
- Retrieval by generic key—a partial key is supplied. Several values can then be used on the OPT= operand to determine the first record retrieved and the direction in which processing will continue: KEQ, KGE and KGT return the first record with a key equal to, equal to or greater than, or greater than the partial key (respectively), and set the retrieval direction to *forwards*, while KEL, KLE and KLT return the highest record with a key equal to, equal to or less than, or less than the partial key (respectively), and set the direction to *backwards*.
- Sequential retrieval—no key is supplied. Records can be returned in ascending or descending key order, commencing with the lowest key or highest key in the file respectively, and continuing forwards or backwards through the file until sequential retrieval is terminated.

Before a record can be retrieved (except for sequential retrieval) a key or partial key must be identified. This is done using the KEY= or KEYVAR= operand on the &FILE GET statement. It is possible to identify the key prior to the &FILE GET (for example, for sequential retrieval) by specifying the KEY= or KEYVAR= operand on a &FILE SET statement.

The success of the &FILE GET statement can be tested using the &FILERC system variable. If the &FILE GET statement was not successful the &VSAMFDBK system variable contains a standard VSAM completion code indicating the type of error that occurred. &FILERC signals end-of-file conditions as well as error conditions.

If generic retrieval or sequential retrieval is performed, the full key of the retrieved record is placed in the &FILEKEY system variable. For example:

```
&FILE OPEN ID=MYFILE FORMAT=DELIMITED
&FILE SET KEY='REC'
.NEXTREC
   &FILE GET OPT=KEQ VARS=A*
   &IF &FILERC EQ 4 &THEN &ENDAFTER &WRITE DATA=END-OF-FILE
   &ELSE &IF &FILERC NE 0 &THEN &ENDAFTER &WRITE +
                DATA=ERROR=&VSAMFDBK
   &WRITE DATA=THE RECORD KEY RETURNED = &FILEKEY
&GOTO .NEXTREC
```

Continuation of a generic retrieval depends on the next &FILE statement. Any statement other than another &FILE GET, destroys the current file position, and ends the generic retrieval. If an &FILE GET statement is issued with a different key from that issued on the first generic &FILE GET, the generic retrieval ends, otherwise the key used on the statement is ignored, and retrieval is determined by the current file position.

Note

When a generic read is performed (for example, when using the OPT=KEQ or OPT=KGE operand on &FILE GET), NCL maintains a generic environment until the NCL procedure specifically stops using generic retrieval, for example by issuing &FILE GET OPT=END. The maintenance of this generic environment holds the current position within the UDB, but in doing so necessarily uses VSAM buffer space. In procedures such as EASINET where there can be much concurrent UDB activity, it is important to keep the length of time that generic environments are open to a minimum and to avoid the use of generic processing where it is not necessary—for example, if &FILEKEY has been set to the full key of a record, do not use &FILE GET OPT=KEQ to read the record.

Restrictions When Using UDBs

When designing facilities that will use file processing the following must be taken into account:

- All records must be keyed, unless processing with an ESDS.
- Keys must range in size from 1 to 255 bytes.
- Keys for base clusters must start in position 0 (unless processing in unmapped mode) of the record. This is often termed *relative key position* (RKP) 0.
- Allowance should be made for field separators, or length and tag bytes added by NCL when determining record sizes for mapped or delimited files.
- The maximum record size is 32K when doing I/O from NCL variables. Otherwise it is the maximum record size for the file.
- When retrieving records, fields are returned in variables which have a maximum length of 256 characters. Creating fields in excess of this length might not be practicable, although a field in excess of 256 characters can span multiple variables after an &FILE GET.
- Unmapped format UDBs have no record structure maintained by NCL. The user is responsible for determining the record format of unmapped format files.
- Mapped and unmapped format UDBs can contain non-character data. The user must make allowance for this and might want to convert the data to expanded hex format after it is retrieved from an unmapped format UDB.
- If a UDB is of a format describable to Mapping Services using a map, then mapped format processing can be performed on it.

Creating UDBs with Alternate Indexes

The description of UDB usage in the preceding sections has been restricted to the processing of records that are identified by a record key starting at position 0 in the record. This is referred to as the *base key* for the file.

The base key allows an NCL procedure to position a UDB to a particular record or group of records within the file, based on the comparison of a key provided as an argument against the keys of records present on the file.

This method of accessing records is adequate for many NCL applications. However, there are other applications which might require access to records within the UDB based on more than one argument.

The use of alternate indexes provides a means by which many arguments can be used to position a UDB to a particular record or record group and allows many different views of the same data held in a single UDB.

As an example, an installation that runs EASINET to provide access to all VTAM applications within its network uses the special LOGPROC NCL procedure to monitor the passing of EASINET controlled terminals to different applications. Each time a logon request is processed, a message is written to the activity log. It is intercepted by LOGPROC and a record written to a UDB that records the time, target application, and name of the terminal concerned.

This information is built up over time and NCL procedures are written to analyze the activity represented by the data on the UDB, where analysis is required by time, by application and by terminal name.

If the base key of the UDB starts with, for example, a time stamp, then the UDB is seen by NCL as being sorted in ascending time stamp order. In this case UDB processing using the time as an argument is very simple; however, if an analysis of the UDB by terminal name is required, the UDB can still be positioned only by the time argument, which is not what is required.

Alternate indices which allow the UDB to be viewed by NCL as being sorted in orders other than by time (for example, by terminal name or by target application name), simplify the processing required in procedures that need to analyze the UDB using arguments that are not satisfied by the base key of the file.

The following sections describe the steps required to create alternate indices for a UDB, the use of alternate indices within NCL procedures and the considerations associated with their use.

VSAM Considerations for Alternate Indexes

An alternate index is an individual VSAM dataset, maintained by VSAM, which contains information and keys to the base cluster VSAM dataset to which the index applies. It can be regarded as providing the user with a different view of the data within the base cluster. As changes are made to records on the base cluster, VSAM (provided that the correct options are defined) automatically updates the alternate indexes associated with the cluster.

VSAM relates an alternate index to its target base cluster with a definition known as a *path*. It is this path which NCL uses to retrieve data using keys from the alternate index.

A given base cluster can have many alternate indices, depending on the complexity of the record formats maintained on the UDB and the number of different ways those records are to be accessed.

To establish a new base cluster with an alternate index do this:

Step 1. Define the base cluster using the IDCAMS DEFINE CLUSTER function.

Step 2. Load the base cluster. This is done using the VSAM IDCAMS utility REPRO statement.

Step 3. Define the alternate index using the IDCAMS DEFINE AIX function.

Step 4. Define the path using the IDCAMS DEFINE PATH function.

Step 5. Build the alternate index using the IDCAMS BLDINDEX function.

Note

This cannot be done for an empty base cluster.

Step 6. Add DD or DLBL statements to the system JCL for both the base cluster and the path (but *not* the alternate index).

Step 7. Assign file IDs to UDBs using the UDBCTL statement.

Note

When defining a base cluster with an alternate index, both the base cluster and the alternate index must be defined with VSAM SHAREOPTIONS of (3 3). This is necessary as two ACBs will be used, one to reference the base cluster and a second to reference the path. A reference to a path causes both the base cluster and its associated alternate index to be opened. VSAM therefore sees two concurrent users of the base cluster.

If duplicate keys are to be allowed on the alternate index, the NONUNIQUEKEY operand must be used when the index is defined. Duplicate keys on the alternate index are likely when the alternate index key is positioned over part of the base key of a cluster.

To establish an alternate index for an existing base cluster, the same procedure is followed but with the omission of the first two steps (that is, defining and loading the base cluster).

The distribution library member DEFAIX contains a sample definition and load of a base cluster with the definition and building of an alternate index and path.

The record key used when processing a base cluster UDB can be 1 to 255 bytes in length and usually starts at offset 0 in the record.

When accessing a base cluster by an alternate index, a key is still used, but the key could be located anywhere within the record that it identifies. The precise location of the key within the record is defined (as an offset from the start of the record) when the alternate index is defined.

Figure 7-2. Base and Alternate Index Keys Within Records

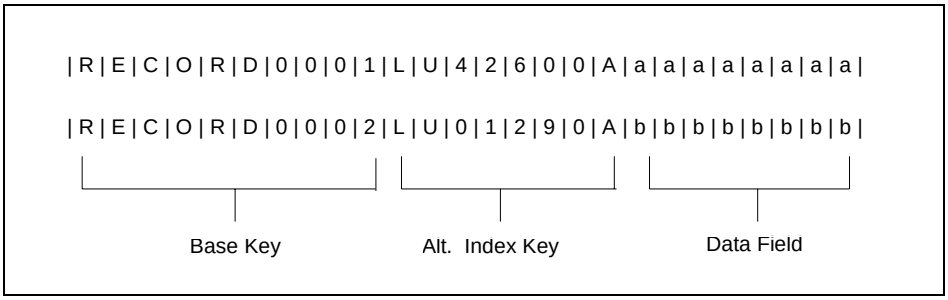


Figure 7-2 shows a representation of two records on a base cluster. The base key of each record starts at offset 0 and is 10 bytes long. In the example the base keys of the two records are:

```
RECORD0001
RECORD0002
```

An alternate index is associated with the base cluster and has been defined with a key that is 8 bytes long and starts at offset 10 in the record, that is, it follows the base key of the record.

In the example the alternate index keys have values of:

```
LU42600A
LU01290A
```

Additional alternate index keys can be defined which span different strings within the base cluster record.

Access to UDBs via an alternate index is similar to standard base key access, but the way the data is presented to the NCL procedure after completion of the &FILE GET is different.

As with standard base cluster usage, the NCL procedure must nominate the UDB to be processed:

- For base cluster processing, the ID operand on the &FILE statement is used to identify the logical file ID associated with the DD or DLBL statement that defines the base cluster dataset in the Management Services JCL.
- For alternate index processing the ID operand on the &FILE statement identifies the logical file ID associated with the DD or DLBL that defines the Path to be used. (Remember that access to a cluster using an alternate index is achieved via a VSAM path definition only and that no direct processing is done on the alternate index cluster itself.)

Figure 7-3. Base and Alternate Index Views of the Same Records

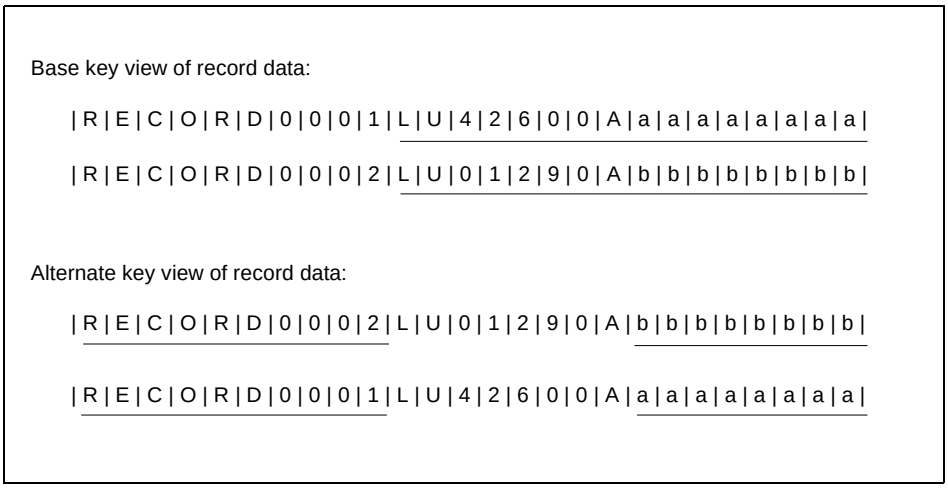


Figure 7-3 shows the different views of the data in two records on a base cluster that are seen when the UDB is accessed through the base key or through the alternate index key. (In each of the example records shown, the key portion is shown in plain text and the data portion is shown underlined.)

Figure 7-4. Reading Records Using the Base Key

```
&FILE OPEN ID=MYBASE FORMAT=DELIMITED                ...access to be by base key
   :
&FILE SET ID=MYBASE KEY='RECORD'                     ...access to be by base key
   :
&FILE GET OPT=KEQ VARS=(1,2,3)                       ...access to be by base key

1st Record on UDB is:

        BASE KEY                        FIELD 1                        FIELD 2
      | R | E | C | O | R | D | 0 | 0 | 0 | 1 | L | U | 4 | 2 | 6 | 0 | 0 | A | a | a | a | a | a | a | a | a |


Result after &FILE GET is:

&FILEKEY=RECORD0001 &1=LU42600A &2=aaaaaaaa &3=null:
   :
&FILE GET ID=MYBASE OPT=KEQ VARS=(1,2,3)             ...read next record



2nd Record on UDB is:

        BASE KEY                        FIELD 1                        FIELD 2
      | R | E | C | O | R | D | 0 | 0 | 0 | 2 | L | U | 0 | 1 | 2 | 9 | 0 | A | b | b | b | b | b | b | b | b |


Result after &FILE GET is:

&FILEKEY=RECORD0002 &1=LU01290A &2=bbbbbbbb &3=null:
```

In the example shown in Figure 7-3, Figure 7-4, and Figure 7-5 the UDB is a UDB (or delimited) format file and each record contains two data fields. The data fields are each terminated by a field separator (X'FF'), although these are not shown in the diagrams. The alternate index key in this example is chosen to overlay exactly the first data field, but does not include the field separator character at the end of the field.

The base key's view of a record on the UDB is that the first 10 bytes of a record are its key and that everything else is data. The alternate index view shows that everything is data, apart from the 8 bytes starting at offset 10 in each record. The effect of these conflicting views on the data actually presented to an NCL procedure on completion of an &FILE GET statement is shown below.

For the example, we will assume that the logical File ID to be used when accessing the base cluster directly is MYBASE and the logical file ID used to access the base cluster via its alternate index is MYPATH.

Figure 7-5. Reading Records Using the Alternate Key

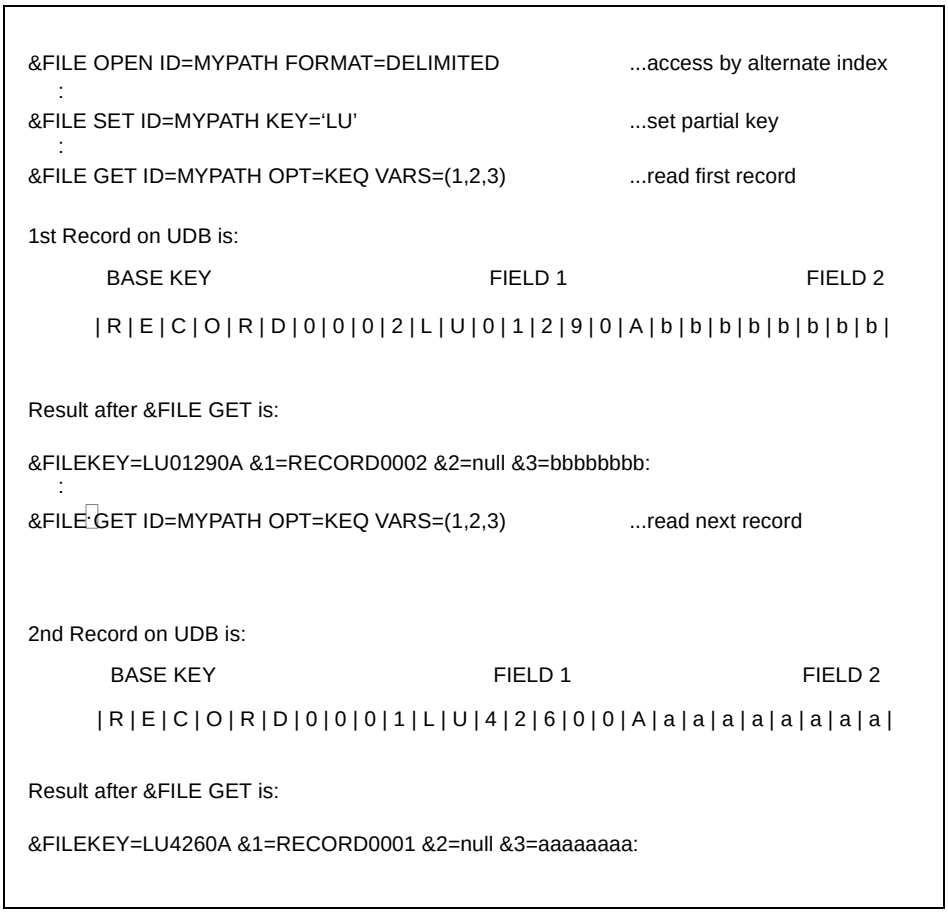


Figure 7-4 shows how data is presented to an NCL procedure that reads records from the example UDB, using the base key to access the file. As noted before, it is assumed here that the alternate key is in fact a separate field within the record, although this does not have to be the case.

Figure 7-5 shows the same sequence of events, but with access to the UDB being via the alternate index. There are two significant differences in the results of the &FILE GET statements compared with using the base key:

- The order of the records is different.
- The data presented in the &FILE GET variables is different.

Records on the UDB are always arranged in ascending key order, according to the key being used to access the UDB.

Therefore, when using the base key the order of the records on this example UDB is:

```
RECORD0001........
RECORD0002........  and so on.
```

However, when viewed from the perspective of the alternate index, the order of the records on the UDB has nothing to do with the value of the base key; as far as the alternate index is concerned the UDB is arranged in ascending order of the alternate keys of the various records on the file. As a result, when accessing records via the alternate index, the order becomes:

```
LU01290A.......
LU42600A.......  and so on.
```

and so, as shown in the figures, the order in which the physical records are retrieved from the UDB depends upon which key is being used to access the file.

The second difference in the result of the &FILE GET statements is explained in the discussion of NCL's handling of keys and record data in the section, *Key and Data Differentiation*.

Controlling UDB Availability

As described earlier, UDBs are made available to NCL procedures through the UDBCTL command which assigns a logical file ID.

For a variety of reasons you might want to change the availability of UDBs from time to time. The UDBCTL command, which is fully documented in your *Management Services Command Reference*, provides the following facilities:

- Stop a UDB:
  This logically blocks all further attempts to access the specified UDB from any procedure, so that no further &FILE statements which specify the ID for this UDB will be allowed. The physical dataset remains open. STOP is rejected if there is any current user of the file.
- Close a UDB:
  Implies a STOP and physically closes the file. This might be necessary if external updating of the UDB is required while the file is still allocated to Management Services.
- Reset a closed UDB:
  In VSAM terms, RESET causes Management Services to open the VSAM ACB with the RST option, which has the effect of emptying the entire file. VSAM constraints mean that RESET cannot be used on UDBs unless they are suballocated (in those levels of VSAM that support suballocation) and is not allowed if the cluster is defined with KEYRANGES or has alternate indexes associated with it.
- Open a UDB:
  Reopens a previously-closed UDB. If the UDB had also been RESET and has not been loaded externally before re-opening, Management Services will initialize the file as described in the section on *UDB Initialization* earlier in this chapter. A UDB that is OPENED requires re-assignment of its logical file ID before it becomes available for processing again.

These UDBCTL options provide operational control over the availability of UDBs to the SOLVE system. UDBCTL can itself be issued from within NCL procedures by suitably-authorized users.

Working with Files

NCL includes an &FILE verb that provides the high-level interface to its file processing facilities. There are seven major operands on the &FILE verb:

&FILE OPEN	Opens a file and identifies its processing mode
&FILE SET	Sets the default file ID, sets the full or partial key, sets the processing mode for subsequent processing
&FILE ADD	Adds a record
&FILE GET	Retrieves a record
&FILE PUT	Adds or updates a record
&FILE DEL	Deletes a current record
&FILE CLOSE	Releases file processing connections

In addition, the following system variables are provided to help test the success of functions:

&FILERC	the completion code for the last function
&FILERCNT	the number of records processed by the last generic delete (&FILE DEL) operation
&VSAMFDBK	the VSAM RPL feedback code from the last function
&FILEKEY	the key of the last record referenced

The &FILE verb and system variables are described in the *Network Control Language Reference* manual.

The OCS mode command UDBCTL, which is used to control the availability of UDBs for processing, is documented in your *Management Services Command Reference.*

Logical File Identifiers

When using NCL, files are referenced by a logical file identifier or *file ID*. The file ID provides a logical connection to the physical dataset. A file ID must be assigned using the UDBCTL command before a file can be referenced by an NCL procedure.

Suppose, for example, that you want to process a file called HELPDESK.DATA, which is defined in the Management Services execution JCL by a DD card with a DD name (or DLBL file name) of HELPDB1 and all NCL procedure references to the physical file are to be made to the symbolic name HELPDESK.

Before the file HELPDESK.DATA can be processed by NCL, a UDBCTL command must be executed specifying the symbolic name by which this file will be referenced from NCL procedures. This UDBCTL command relates the DD name or DLBL file name of the dataset with its symbolic name and is coded:

```
UDBCTL HELPDB1=HELPDESK
```

Alternatively, a file ID can be assigned using the ID operand of the UDBCTL command when it is opened:

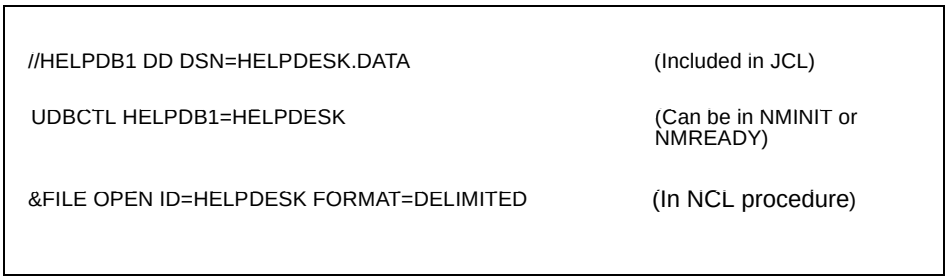
```
UDBCTL OPEN=HELPDB1 ID=HELPDESK options
```

For a particular NCL procedure to access the file that has now been assigned the symbolic, or logical name of HELPDESK, the procedure must contain an &FILE OPEN statement specifying the file that is to be processed:

```
&FILE OPEN ID=HELPDESK FORMAT=DELIMITED
```

The relationship between the DD name (or DLBL file name), UDBCTL command and file ID is summarized in Figure 7-6.

Figure 7-6. Relationship Between DDNAME, UDBCTL, and &FILE OPEN



This approach offers considerable flexibility when it becomes necessary to change the system configuration, since all NCL procedures reference files using their logical file ID and are therefore shielded from external changes.

Should a dataset become full, a single UDBCTL command can simultaneously migrate all NCL procedures to reference a new dataset without change to the NCL procedures. Consider our example using the HELPDESK file. It might be desirable at the end of the day to swap to another dataset and perform other processing on the previous day's data while continuing to record new problems in another file. Multiple datasets would be included in the system JCL and a

UDBCTL command used at the appropriate time to swap to an alternative dataset. For example:

```
UDBCTL STOP=HELPUDB1(stop current use)
UDBCTL HELPUDB2=HELPDESK(swap to new file)
```

The NCL procedures continue to use the same ID on the &FILE statement, but now use a different physical dataset as the UDBCTL command has changed the logical relationship:

```
&FILE OPEN ID=HELPDESK FORMAT=DELIMITED
(NCL procedure is unchanged)
```

Releasing File Processing Resources

The &FILE OPEN statement allocates certain resources to the requesting NCL procedure. It is not normally necessary to release file processing resources within an NCL procedure. This is performed automatically when the NCL procedure terminates.

Under certain circumstances, such as in an EASINET procedure, where there might be many concurrent users performing file processing, it might be desirable to release any file processing overheads when they are no longer required, to ensure that system overheads are minimized.

This can be accomplished using the &FILE CLOSE statement. &FILE CLOSE allows either specific files or all files to be freed. When this is done, any storage associated with processing the file is released and the connection is logically severed for that user.

Having used &FILE CLOSE to release a particular file, connection can be re-established using another &FILE OPEN statement.

&FILE CLOSE destroys any generic retrieval position a user might have established within a file and any subsequent reference would have to reestablish that position if required.

Displaying File Information

The SHOW UDB command can be used to display details about files available to the system. This information includes details about the number of active users, space utilization and the current status. In addition, any open error codes that caused a file to be disabled is displayed.

The SHOW VSAM command is used to display the VSAM attributes of the various VSAM datasets in use by Management Services. Information presented by this display includes:

- Record sizes
- Control Interval sizes
- Control Interval and Control Area split statistics
- Current index and data buffer allocations
- Information on string and buffer shortages that might have occurred
- LSR pool statistics (in OS/VS systems) .

The SHOW UDBUSER command is used to obtain information on current usage of particular UDBs in the system. The display lists all UDBs showing DD or DLBL names, file IDs and the names of the user IDs and NCL procedures that are currently accessing each UDB.

The SHOW UDB, SHOW VSAM, and SHOW UDBUSER commands are described in your *Management Services Command Reference.*

Specifying the File Processing Mode

The &FILE statement has a mandatory ID=*fileid* operand. This operand identifies the UDB that the NCL verb is to be actioned on. The ID operand enables several files to be open and processed simultaneously. There is also a FORMAT operand on the &FILE verb which can only be specified in conjunction with the OPEN or SET operands on the file verb (that is, &FILE OPEN ID=... FORMAT=... and &FILE SET ID=... FORMAT=...). The FORMAT operand is used to specify the current processing mode for a file (delimited, unmapped, or mapped). It is possible to switch between two file formats without losing position.

Specifying the File Key

There are several ways in which a file key can be specified on an &FILE statement. The way in which the value is specified on the KEY= operand is independent of the current processing mode (delimited, mapped, or unmapped). The value of the KEY= operand is interpreted as a character string by default. The value can be specified inside quotes, and substitution is carried out on any variables contained inside the quotes. You can put the letter X after the quotes to indicate that the data between the quotes is to be hex-packed to create the key.

For example:

```
KEY='&A 0001'
```

where &A=AAAA indicates a key of: 'AAAA 0001'.

```
KEY='C1C1C1C140F0F0F0F1'X
```

also indicates a key of 'AAAA 0001'

There is also a KEYVAR= operand on the &FILE verb, which is an alternative to the KEY= operand. It is used to specify a variable whose contents will be used as the key. For example, `KEYVAR=A` indicates that the contents of the variable &A will be used as the key. In this case, if &A contains non-printable characters or trailing blanks, they appear in the key unchanged.

Working with Data

As with other NCL functions, data manipulated by file processing is usually maintained in tokenized form (for example, as NCL variables). The &FILE verb also enables data to be maintained in an MDO, but only for mapped format file processing. A typical NCL procedure might accept input from a full-screen panel in a series of tokens or variables and then add this data to a UDB. Alternately, it can receive data from another application using APPC into an MDO, and then add this data to a mapped format UDB.

The physical representation of the data on the UDB depends on its format:

- For mapped format files using $NCL map, the record consists of a list of vectors. Each vector has a two byte length field followed by a two byte tag which is followed by the data from the token. The entire list is encapsulated with its own two byte length and tag fields. See the distributed $NCL map for details of the data structure and tag values.
- For unmapped format files, the file processing statements still use variables, but NCL does not regard each variable as a field within the record. When writing a record to an unmapped format UDB all variables are written contiguously (that is, no field separators are inserted). The structure of the data within the record is therefore the responsibility of the user.
- For delimited format files, each variable is treated by NCL as a field within a record, each field being separated from its neighbor by a field separator. When data is retrieved it is returned in a series of variables using the field separators to determine the size of each field. This approach relieves the procedure of having to define the format and structure of data in UDBs.

When data is retrieved from an unmapped format UDB, NCL again makes no attempt to identify individual fields within the record. The entire record is read from the UDB and then split into as many variables as are necessary to hold the data that has been read. Alternatively, the user can indicate how many bytes of data from the record are to be placed into each variable on the &FILE GET statement.

When data is retrieved from a mapped format UDB it is placed into an MDO unchanged from the way it existed on the file. It is also possible to specify the name of a map on the file verb which will be attached to MDO when data is placed into it. See Chapter 8, *Using Mapping Services*, for more information on MDOs and maps. When writing to a file using mapped format, the contents of the MDO are simply placed on the file as they are.

A more detailed discussion of the format of data within variables is provided later in this chapter.

Dataset Positioning and Generic Retrieval

NCL supports both sequential and generic retrieval from keyed datasets. Such functions imply that a current position within the file is maintained so an NCL procedure can simply request the next record and it will be supplied. No incrementing of keys by the NCL procedure is necessary.

Under certain circumstances, such as with generic retrieval, it might be necessary to alter the retrieval sequence and commence retrieval using a different key.

NCL must be informed that such a change is required and that the current retrieval sequence is to be stopped. This is done using the &FILE GET ID=*name* OPT=END statement. This indicates to NCL that generic retrieval is to be terminated in anticipation of some other processing.

If an end-of-file condition is signalled, no &FILE GET ID=*name* OPT=END is required. The use of a non-generic function, such as the specific retrieval of a record, or the use of the KEY= or KEYVAR= operand on the &FILE verb to set the current key, will also cancel a previous generic function.

You can use KEY=‘ ’ and the generic option together, in which case the key is used to gain initial position for the generic retrieval. Subsequent generic retrieval requests continue from this position, as long as the key supplied (if any) is the same as the initial key.

Mapped Format Files: Data Representation

When a file is processed using the mapped format processing option, a map is used to describe the arrangement of the records. The map describes the records in terms of structures or components. Maps are managed by Mapping Services and exist as separate entities within Management Services. When carrying out a retrieval using mapped format processing, the contents of the file record are usually placed into an MDO, and a map is then used to interpret the contents of the MDO. A full description of how to use MDOs, maps, and Mapping Services to manipulate MDOs is provided in the chapter titled *Using Mapping Services* in this manual.

It is also possible to read and write NCL tokens using mapped format processing. This is a special case of mapped format processing and a special system map called $NCL is used for this purpose. When NCL variables are written to a file using mapped format processing, and the $NCL map, they appear on the file as subvectors. The subvectors consist of a header followed by data. There is one subvector for each token written to the file. The subvector headers consist of a 2 byte length field followed by a 2 byte key where the key is X'0000'. (For example, a variable containing the value X'C1C1' would appear on a file record as: X'00060000C1C1', where 0006 is the length, 0000 is the key, and C1C1 is the data.)

The process of writing variables to a mapped format file using the $NCL map is reversible. This means that if a set of variables is written to a record under these conditions, and the contents of this record are then read back into the variables at some other stage, the original values contained in the variables will be the restored.

Note

This allows tokens containing non-printable characters to be handled, that is, data transparent.

Unmapped Format Files: Data Representation

Unmapped format UDBs, which are usually files created or processed by mechanisms other than NCL, can contain non-character hexadecimal data (for example, records might contain strings of binary zeroes). When data is read from a file using unmapped mode, the entire record is read in byte-for-byte *as is*, and non-printables remain as they appeared on the file. The data is placed into variables in lots of 256 bytes (the maximum variable size) unless otherwise specified. If the data contains non-printables, it is advisable to hex expand the data using the &HEXEXP statement before processing it. It is possible to specify a length of 128 for each variable on the &FILE GET statement, so that no overflow occurs when the data is hex expanded.

Note

Most data formats are most conveniently processed using mapped format files and maps developed using Mapping Services.

When writing to an unmapped file, the contents of the output buffer are placed onto the file unchanged, including non-printables. If VARS= or ARGS is specified on the &FILE PUT/ADD statement, the contents of each of the variables are concatenated and placed onto the file. Any non-printables that were contained in the variables will also be placed onto the file. If DATA= is specified on the &FILE PUT/ADD, substitution takes place on the buffer before it is written to the file.

If any of the variables specified following the DATA= operand contain non-printables, these are preserved. Spaces and text between variables is also written to the file.

For example, if the variable &A contains X'C100C1' and the variable &B contains X'C200C2' then after the following statement is executed:

```
&FILE PUT ID=MYFILE DATA=&A XXX &B
```

the following will appear on the file:

```
X'C100C140E7E7E740C200C2'
```

One of the principal uses for unmapped format processing is in the dynamic preparation of reports from NCL procedures (particularly in MVS systems), where an ESDS UDB is allocated dynamically to SYSOUT, report lines are written in unmapped format and the UDB is then closed for immediate printing.

Note that when using &FILE PUT/ADD to write to SYSOUT file, carriage control options are available to assist with report formatting. See the description of the PUT and ADD operands on the &FILE verb.

DBCS Considerations When Using Files

If you write data to a UDB that represents a DBCS datastream, the datastream is written to the file using shift characters that depend on the DBCS mode in which Management Services is operating. This is determined by the SYSPARMS DBCS= operand. A system that reads that file must therefore operate in the same DBCS mode as the system that originally wrote the data, otherwise the reading system will not be able to recognize the shift characters present in the data.

See the chapter titled *Customizing Management Services* in the *Management Services Implementation and Administration Guide* for details.

&FILE GET Statement and Unmapped Format UDBs

The &FILE GET statement is used to retrieve data from any type of UDB and when issued causes NCL to present data to the NCL procedure in a series of variables, or as an MDO.

It is only possible to use the MDO= operand for mapped format processing. As described earlier, if the UDB is a delimited format file, each variable on the &FILE GET statement is returned with the contents of field, where the fields are segments in the record delimited by X'FF'.

However, if the UDB is an unmapped format UDB, NCL cannot provide the data to the procedure as a set of individual fields—NCL has no knowledge of field boundaries within the record.

Therefore, for unmapped format UDBs, NCL treats the record read from a UDB as a single string and then splits this string into as many full-length variables as are required to hold all the data, or as many as are provided on the &FILE GET statement, whichever is the smaller number.

A full-length variable is 256 characters long. It is also possible to indicate explicitly how many bytes are to be placed into each variable.

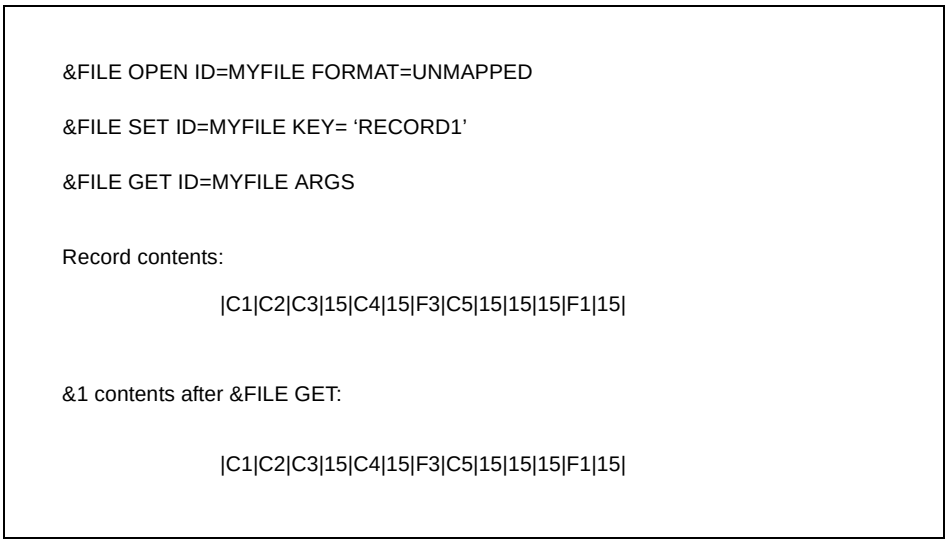
For example:

```
&FILE GET ID=MYFILE VARS=(A(10),B(100),C(40))
```

indicates that the first 10 contiguous bytes of the record being read will be placed unchanged into the variable &A, the next 100 bytes will be placed into variable &B, the next 40 bytes will be placed into variable &C, and any remaining bytes will not be placed into any variable (that is, they will be ignored).

Figure 7-7 shows an example of using &FILE GET on an unmapped format file.

Figure 7-7. Reading UNMAPPED Data into a Variable



Data Conversion and Unmapped Format UDBs

If data on an unmapped format UDB is expected to contain non-printable characters it is wise after reading the data into variables to convert it to an expanded hexadecimal form in order to preserve the non-printables (this is because many NCL verbs automatically translate non-printables to blanks—X'40's).

Regardless of the data representation format selected on the &FILE OPEN/SET statement, data read from or written to an unmapped format UDB might require conversion from expanded hexadecimal format to character format, or conversion from character to expanded hexadecimal format. Two NCL statements, &HEXPACK and &HEXEXP provide this facility.

For example, where &HEXEXP is being used for file processing and data is being read from a file which has records which consist of a 1 byte error code followed by text. If the error code is not X'00', then there is a problem with the record.

A record could be read from the file as follows:

```
&FILE OPEN ID=MSGFILE FORMAT=UNMAPPED
&FILE GET ID=MSGFILE KEY='00000001'X
VARS=(ERRCODE(1),MSGTEXT)
&TEMP = &HEXEXP &ERRCODE
&IF &TEMP GT 00 &THEN &GOTO .ERROR
    .
    .
    .
&WRITE DATA=FOLLOWING MESSAGE READ FROM FILE: &MSGTEXT
```

The record with a key of X'00000001' was read into two variables, &ERRCODE and &MSGTEXT. A subscript was used to indicate that the first byte was to go into &ERRCODE, and the remaining bytes (up to 256) were to be placed into the variable &MSGTEXT.

An &HEXEXP statement was used to convert the &ERRCODE variable from binary (Packed Hex) format into NCL format. (For example, if &ERRCODE was X'00', &HEXEXP would convert it to X'F0F0'). This enabled the value of it to be tested for an error condition.

The functions and syntax of the &HEXPACK and &HEXEXP verbs are described in detail in the *Network Control Language Reference*.

Note also that a map could have been used to describe the above mentioned file, and then the record could have been read into an MDO using mapped format. See the chapter titled *Using Mapping Services* for more information on using mapped format files.

Key and Data Differentiation

When a record is read from the UDB, whether it is a delimited format or an unmapped format file, NCL normally performs the following steps when preparing the data for presentation to the procedure, regardless of the key under which the record has been retrieved.

1. The record is read as a single string.
2. The full key of the record is extracted and placed in &FILEKEY.
3. If the file being processed is a delimited format file and the record is not read under the base key, the section of the record previously occupied by the key is replaced by a field separator (X'FF').
4. The remaining data is placed into the variables nominated on the &FILE GET statement according to the rules applicable to the type of UDB being processed.

The significant point here is that the key under which the record is retrieved is not, by default, regarded by NCL as part of the record data and does not therefore appear in the &FILE GET variables.

The result of this is that the same fields within a given record can be relocated into different &FILE GET variables, depending upon which key is used to read the record.

Figure 7-4 shows that when the records in the example are read by successive &FILE GET statements, the two separate fields of which the records are composed are placed into the two variables &1 and &2, as expected. &3, specified on the &FILE GET statement, is set to null since no data was available to place into it.

However, Figure 7-5 shows a different &FILE GET result when the alternate index is used to read the same two records. NCL has obeyed the rules for data presentation with the following result:

- The record is read under the alternate index key.
- The key used (LU01290A) is extracted and placed in &FILEKEY and replaced with a X'FF' field separator. Remember that the key in this case is followed by a field separator too, so now there are two consecutive field separators in the record.
- The data is placed into the nominated variables field-by-field, starting at the left-hand end of the record. This results in values:

```
&1 = RECORD0002
&2 = null
&3 = bbbbbbbb
```

The reason that &2 is null is because that which used to be the key value has been extracted from the record and replaced with a field separator. As NCL scans the data to determine the location of the various fields, two consecutive field separators are found indicating the presence of a null, or zero-length field. Consequently, &2 becomes a null variable. The result of this is that the data (*bbbbbbbb*) is no longer located in the same variable as it was when the record was read under the base key.

Note

When processing with alternate indices the positioning of the alternate key can significantly impact the way in which an NCL procedure must process the UDB. It is therefore recommended that you familiarize yourself with this process and try a few simple examples first, to ensure you understand it.

Key Extraction Options

Although the rules for data presentation allow accurate prediction of what each &FILE GET variable will contain regardless of which key is used to retrieve a record, the effects of key extraction from record data can sometimes prevent the use of common processing logic when the same procedure uses different keys to access the same UDB.

The &CONTROL option NOKEYXTR lets you specify that NCL is to leave the key under which a record is read in the data portion of the record (as well as placing a copy of the key in &FILEKEY as usual). This option applies only when reading data via a key that does not start at offset 0 within the record.

The effect of NOKEYXTR is to eliminate null variables occurring after an &FILE GET (where the null represents the original location of the key). NOKEYXTR also eliminates the splitting of single fields into two when the key happens to overlay part of a single field.

We recommend you use NOKEYXTR in those procedures that reference a UDB under multiple keys where you wish to provide common processing routines for specific fields within the records, regardless of the key under which the records are read.

Update Restrictions on Alternate Indexes

NCL supports the retrieval of data across any number of alternate indexes. However, data updating is restricted to the base cluster for delimited format files. If a record that has been retrieved using an alternate index is to be updated, the ID operand on the &FILE statement is used to indicate that processing has swapped to the base cluster, and that the necessary key must be set (either on an &FILE SET, or on the &FILE PUT), before the record can be written back to the UDB.

Note that even when using the base cluster, the base key must start at offset 0 within the record (RKP 0) for a UDB (or DELIMITED) format file. No updating is allowed for any cluster which violates this rule.

This restriction does not apply to unmapped format UDBs which can be updated even if the key does not start at offset 0. In this case if insufficient data is supplied to the left of the start of the key, null padding (X‘00’) is added up to the start of the key location.

Off-line Processing of Datasets

In MVS systems the DEALLOCATE command can be used to release a UDB for off-line processing. Before deallocation, the UDB must be closed using the UDBCTL command.

This can only be performed once all current users have ceased using the UDB. If deallocation cannot be used, the stripping of files created by NCL for further off-line processing can be achieved without a restart of the system if the following rules are followed:

- VSAM SHAREOPTIONS must allow concurrent access to the dataset.
- DISP=SHR must be specified on the dataset (OS/VS).
- The UDBCTL CLOSE=*xxxxx* operand is used to stop further logical connections to the file and to close the physical dataset. This is allowed only if there are no users currently referencing the file.
- Use a utility to strip the file (for example, the VSAM REPRO utility). Any off-line processing programs must take any high-value (X‘FF’) field separators into account when determining record formats on UDB format files.

Backing Up Online Datasets

While standard batch utilities can be used to backup off-line VSAM datasets they cannot be safely used to backup active online datasets.

Only a dataset that is closed and preferably deallocated from Management Services should be backed-up by an off-line utility.

If an off-line utility is used to backup an active on-line dataset, the backup will complete as though it were successful. A later restoration of that dataset could result in a corrupted file. This happens because the off-line utility has no knowledge of the current state of the file or any in-storage indexes or buffers that could have been written to the file at the time the backup is performed. To resolve this problem, perform on-line backups of active datasets. The system is supplied with a utility (UTIL0007) which invokes the VSAM IDCAMS utility on-line to perform a backup. In operating in this manner the on-line utility has access to the in storage buffers and indexes and can guarantee a usable backup of the dataset.

For operational reasons it might still be desirable to temporarily halt activity on a file so that a precise recovery time is known.

8

Using Mapping Services

This chapter discusses the following topics:

- What Is Mapping Services?
- Mapping Services Processing
- Mapping Concepts
- Using MDOs and Maps in NCL

What Is Mapping Services?

Mapping Services is a SOLVE feature which provides NCL with enhanced capabilities for manipulating data with various formats.

Early versions of NCL stored program data only as *tokens* (or NCL variables). Mapping Services enhances NCL by introducing *Mapped Data Objects* (MDOs) as an alternative to tokens and as a means of storing data. Data can easily be transferred between MDOs and tokens. MDOs can also be written directly to files and are supported by a number of specialized NCL functions such as &ASSIGN and &APPC. MDOs can be used in many situations where tokens are used.

An MDO can contain any data that can be represented as a continuous string of bytes in storage. The primary advantage of an MDO over NCL tokens is that particular substructures within an MDO can be located and referenced by name from NCL, using various rules which Mapping Services understands. Various segments within these substructures can also be referenced. The substructures which can be referenced within MDO data are generically known as *components*.

To distinguish between various substructures which exist within a particular MDO, a unique tag or key is assigned to each component. To determine what names are to be associated with particular components within an MDO, and their tag values, a *map* is used. A map enables the NCL language to associate names with the various components which exist within a particular MDO. Thus a map provides NCL with the ability to interpret the data contained within an MDO.

Mapping Services Processing

Mapping Services extends the flexibility of NCL to include the manipulation of data in any format. There are three key components involved in Mapping Services processing:

- MDOs
- Maps
- NCL procedures

Mapping Services provides a set of facilities that effectively mediates between these three components and manages them. This allows components to exist as separate entities, which interact during NCL processing to perform data manipulation.

MDOs

Mapping Services can operate on any data item that can be represented as a continuous string of bytes in storage. In addition, Mapping Services understands certain rules which let it locate substructures within those data items if they are composed of variable data structures.

To assist with uniformity of processing, Mapping Services treats the entire data item as a structure called an MDO. An MDO has a name that is supplied by the NCL procedure to reference that instance of data.

Maps

Maps are defined using the Abstract Syntax Notation One (ASN.1) language. They are then compiled and loaded for use in NCL.

For more information on defining maps, see the chapter titled *Using Mapping Services in NCL* in your *MODS Programming and Administration Guide.*

NCL Procedures

NCL procedures are written to access data through one or more of the standard NCL verbs. When data is provided by one of the verbs that support Mapping Services, the procedure can request that it be treated as an MDO. It provides a name for the MDO, and (optionally) the name of a map that can be used to interpret the data.

Once the data has been internally accessed, and before the NCL verb completes, Mapping Services makes a connection between the MDO and the designated map. The NCL procedure can then reference data components within the MDO using the symbolic names defined in the map.

A simple example of how the three components interact in NCL is shown below:

```
&ASSIGN MDO=file1rec MAP=file1map
                  -* the map called file1map is attached
                  -* to the MDO called file1rec.
                  -* This statement indicates to Mapping -*
Services that this map is to be used -* to interpret any
data in the MDO.

&FILE OPEN ID=FILE1 FORMAT=MAPPED MAP=file1map
                  -* Open file in mapped processing mode.

&FILE GET ID=FILE1 KEY='00000000' MDO=file1rec
                  -* Read a record from the
                  -* file into the MDO.
```

```
&ASSIGN VARS=A FROM MDO=file1rec.luname
                  -* Locate a component within the MDO
                  -* data and copy its contents into the
-* variable &A.
                  -* Mapping Services uses the map to find
-* out how to locate and recognize the -* structure
referred to as luname    -* within NCL.
```

Mapping Concepts

A map effectively describes the following to NCL:

- The names that will be used to reference various components within an MDO
- The relationship between these data components
- The means of identifying each component
- The way in which the data is represented

Until a map is associated with an MDO, NCL cannot reference components in the data by name. However, NCL can still process the MDO as a whole.

Using MDOs and Maps in NCL

Data Sources

NCL processing can move data directly into an MDO using many verbs, via the following sources:

- From files
- Across APPC transactions
- From NCL tokens (as variables and arguments, or args)
- Across PPI transactions
- Using &INTREAD, &LOGREAD, &MSGREAD, &PPOREAD
- From vartables
- From CNM
- Using BER encode/decode

Naming

It is possible to reference structures within MDO data by name from NCL. A map must be attached to the MDO so NCL can use this map to associate names with physical structures which can occur within the data. By definition, components are hierarchically arranged. This means that to reference one component enclosed within another, a concatenation of the enclosing structure names is required to identify the enclosed one.

For example, the name:

```
MDO=FILE1REC.DOMAIN.SUBDOMAIN.LU
```

might be used to reference the structure LU, within a structure called SUBDOMAIN, which is contained in a structure called DOMAIN, which is within the data in an MDO called FILE1REC.

To reference all data contained within an MDO, only specify the single MDO name segment, as follows:

```
MDO=FILE1REC
```

Transferring MDOs Between Nested NCL Procedures

It is possible to share an MDO between nested procedures by using SHRVARS options. Any MDO selected by the current SHRVARS option is shared in the same manner that NCL tokens are shared.

For example:

```
&ASSIGN MDO=ABC MAP=MAP1 DATA=xxxx
                  -* Create an MDO called ABC, attach the
                  -* map called MAP1 to it, and assign
                  -* data into it.
&ASSIGN MDO=XYZ MAP=MAP2 DATA=yyyy
                  -* Create an MDO called XYZ
&CONTROL SHRVARS=(A)
                  -* Set SHRVARS option.
-EXEC PROC2       -* Call procedure PROC2.
```

In the above example, the MDO called ABC is available in the nested procedure but the MDO called XYZ is not.

There is no support for passing an MDO as an invoked or returned parameter on nested calls. Only NCL tokens can be passed in this manner. However, MDOs can be transferred from one NCL process to another using &WRITE and &INTREAD or any of a number of other verbs.

Mapping Services Mapping Support and NCL Processing

Before structures can be referenced within an MDO by name from NCL, a map must be assigned to the MDO. The map provides NCL with the following information about an MDO:

- The type of structures which can be expected within the MDO data
- How to locate and recognize these structures
- The names which will be used in NCL to reference these structures

Maps defined to Mapping Services are loaded automatically the first time they are referenced from NCL. They can also be loaded manually by entering the SYSPARMS MAPLOAD=*mapname* command.

The MAPLOAD command will only load a map not already in memory. To reset a map in memory (for example, after modifying an existing map), use the SYSPARMS MAPRESET=*mapname*. It will be automatically reloaded on demand.

Connection to Mapping Support

An MDO is usually attached to mapping support at the same time it is created or data is first placed into it. The MAP= operand is available on many verbs and is used to attach a map to an MDO.

For example:

```
&ASSIGN MDO=xxx MAP=MYMAP
         -* This creates the MDO if it didn't already
         -* exist, and assigns a map to it.

&FILE GET ID=FILEID MDO=yyy MAP=MYMAP2
         -* This creates the MDO if it didn't
         -* exist, places data from the file record into -*
it, and attaches the map to it.

&APPC SEND...  VARS=a* MAP=MYMAP2
         -* This transmits the data in the variables, and
         -* the mapname.
&APPC RECEIVE...  MDO=zzz
         -* This receives the data into the MDO, and
         -* automatically attaches any map received to the
         -* MDO as well.
```

Sourcing Data

MDO data can be sourced from NCL tokens, many NCL input verbs or as a hard-coded string. An MDO does not have to be attached to a map before data can be placed into it, because MDOs and maps are separate entities in Management Services. It is possible to attach a map to an MDO before or after data has been placed into it.

For example:

```
&ASSIGN MDO=ccc DATA=data
                          -* assign data into the MDO
&ASSIGN MDO=ccc MAP=mapname
                          -* attach a map to the MDO.
```

Note

After map connection, the &ZMDORC system variable should be checked to ensure the connection was good. If Mapping Services detects a mismatch between the map definition and the MDO data, &ZMDORC contains a non-zero value and subsequent access to the MDO can produce name checks or type checks.

Manipulating and Extracting Data

Data in an MDO is manipulated using the &ASSIGN statement. There are two major operand forms on the &ASSIGN statement; an MDO *stem* name, and a *compound* name.

Stem Name: MDO=*vvv*

Compound Name: MDO=*aaa.bbb.ccc*

The stem name is used to refer to the entire data portion of an MDO. An MDO does not have to have a map attached to it to be referenced by its stem.

The compound name is used to refer to a structure within the MDO. A map must be attached to the MDO for this type of reference, otherwise NCL will not be able to locate the structure. (NCL cannot make sense of the data without a map.)

The &ASSIGN verb is useful when extracting individual components in an MDO into NCL tokens.

Using a Map in NCL Processing

After a map has been defined, it can be used to interpret the contents of MDOs in NCL.

The following example demonstrates how MDOs can be used for the encapsulation and transmission of data in NCL.

Procedure A can receive messages from procedure B, and perform some desired action on some of the data in the message. If the action is completed successfully, procedure A will return the modified data to the initial sender (procedure B), with a message indicating the operation was successful. If the action is not completed successfully, procedure A will carry out some extra error processing and then return a message to the sender indicating that the operation was unsuccessful.

The message data is mapped with a map that contains the following components:

Component	Purpose	ASN.1 Type
ACTION	The action to be performed	GraphicString
DATA	The data on which to perform the action	OCTET STRING
ACTIONRESULT	The result of the action processing	ENUMERATED

Procedure A could be as follows:

```
.RECEIVELOOP
    &INTREAD MDO=ACTIONPARMS      -* Read the MDO from the sender
    &ASSIGN VARS=SENDER FROM MDO=$INT.SOURCE.NCLID
                                  -* Extract the senders NCLID
                                  -* from the system MDO, $INT,
                                  -* which is always set after an
                                  -* &INTREAD operation, and is
                                  -* mapped by the $MSG system map
    &ASSIGN VARS=ACTION FROM MDO=ACTIONPARMS.ACTION
                                  -* Extract the type of
                                  -* action to be performed
    &ASSIGN VARS=ACTIONDATA FROM MDO=ACTIONPARMS.DATA
                                  -* Extract the data on which
                                  -* to perform the action
    -EXEC ACTPROC ACTION=&ACTION ACTIONDATA=&ACTIONDATA
                                  -* Call proc to carry out
                                  -* action
    &IF &RETCODE EQ 0 &THEN +     -* Check return code from action
       &DO
          &ASSIGN MDO=ACTIONPARMS.DATA FROM VARS=ACTIONDATA
                                  -* Set modified data in MDO
          &ASSIGN MDO=ACTIONPARMS.ACTIONRESULT DATA=OK
                                  -* Set result of action in MDO
          &WRITE NCLID=&SENDER MDO=ACTIONPARMS
                                  -* Return modified MDO to caller
       &DOEND
    &ELSE +
       &DO
          &GOSUB .ERRORPROC       -* Do error processing
          &ASSIGN MDO=ACTIONPARMS.ACTIONRESULT DATA=FAIL
                                  -* Set results of action in MDO
          &WRITE NCLID=&SENDER MDO=ACTIONPARMS
                                  -* Return MDO to caller
       &DOEND
&GOTO .RECEIVELOOP                -* Loop to process next message
```

You can see that Mapping Services greatly simplifies the NCL required to process complicated data formats. Often, externally-sourced data has a format which is awkward to manage in NCL. In these situations, Mapping Services can prove to be a very useful tool.

9

Using Mapping Services with Management Services

This chapter discusses the following topics:

- Overview
- MDO Behavior and NCL Processing Conventions
- Using the &ASSIGN Verb
- Querying MDO Components
- NCL Reference, Type Checking, and Data Behavior
- Type Conversion for MDO Assignment

Overview

Mapping Services uses the ISO standard Abstract Syntax Notation One, or ASN.1, as the map definition language. The added sophistication that ASN.1 brings to the definition phase leads to greatly improved application use of Mapped Data Objects (MDOs).

By understanding ASN.1 and how Mapping Services implements its concepts, significant advantages in NCL processing of complex data structures can be realized. Using ASN.1 in defining maps, all MDO components have a data *type*. Only data that is of the correct type for the component can be assigned into a component. An attempt to place an invalid value into an MDO component fails with a *type check* return code.

There are two types of data:

- Simple - for simple data types the assigned data should be of the correct format for the component type, otherwise a type check results.
- Constructed - for constructed data types, entire construction must be valid according to both the logical type (as described by the ASN.1 type definition), and also the physical type (as described by any Mapping Services implementation-specific definitions), otherwise a type check or data check results.

MDO Behavior and NCL Processing Conventions

After using any verb that references an MDO, the MDO return code (&ZMDORC) and feedback (&ZMDOFDBK) system variables are set. Therefore you should check most verbs when using operations involving MDOs.

Since &ASSIGN is used more frequently to process MDO data, an alternative exists. If &CONTROL MDOCHK is in effect, any error situations that normally result in a return code of 8 or higher cause the NCL procedure to terminate, but only if the return code or feedback variable is set by &ASSIGN.

If &FILE GET MDO=*xxx* sets &ZMDORC to greater than 8, the process will not terminate if &CONTROL MDOCHK is in effect. However, if the default &CONTROL NOMDOCHK is in effect , all error checks are reported via the return code and feedback values.

The possible values of the return code and feedback system variables and their meanings are shown in the following table.

&ZMDORC	&ZMDOFDBK	Meaning
0	0	ok
4	0	null: optional component present but empty, or null data assigned to optional component
	1	null: optional component not present
	2	null: mandatory component present but empty, or null data assigned to mandatory component
	3	null: mandatory component not present
	4	string was truncated (applies to FIX offset or length components only)
8	0	type check: data is invalid for type
	1	data check: data is invalid structurally —a common cause is data too long or too short
12	0	name check: component not defined
	1	name check: index position invalid or value is out of range
16	0	map check: map not found
	1	map check: map contains errors, load failed
	2	map check: map/data mismatch

To minimize the use of these return codes it is necessary to understand the general behavior of MDOs with NCL.

An &ZMDORC value of 0 means that the data referenced was of a valid type for the component referenced. On assignment the data will be formatted and placed in the target component, resulting in an &ZMDOFDBK value of 0.

An &ZMDORC value of 4 is returned when an MDO component has a null, or empty, value (unless it was the NULL type in which case &ZMDORC is 0). Any component, regardless of its type, can be set to a null value. &ZMDORC is also set to 4 when a string type is truncated and an &ZMDOFDBK value of 4 is returned, but only if it is a fixed length component.

An &ZMDORC value of 8 is returned if &CONTROL NOMDOCHK is in effect and the data referenced does not conform to the type of the component referenced, or cannot be assigned for other reasons. In exceptional circumstances a type check or data check might result on a read intent operation. More usually however, it occurs on an update intent operation where the data being assigned is invalid for the data type and a type check results. If on an update operation the function cannot be performed for other reasons, usually due to insufficient available space in a component that cannot be further extended, a data check will result. In all such cases the operation fails, and the referenced MDO component is unchanged.

An &ZMDORC value of 12 is returned under the following conditions:

- &CONTROL NOMDOCHK is in effect and the MDO is mapped but the specific component referenced was not defined in the map
- The name is valid, but an index was used on a component that is not allowed to be indexed
- The index exceeds the defined index range

An &ZMDORC value of 16 is returned if, on any verb, a map connection request fails. &ZMDOFDBK indicates the reason that the map connection failed.

The convention in using these return code system variables is discussed further below. Other system variables available to interrogate error conditions are:

&ZMDOID
Identifier of last *known* MDO involved in last operation.

&ZMDOMAP
Map name for &ZMDOID.

&ZMDONAME
Fully qualified name of the MDO component involved in the last operation.

&ZMDOCOMP
Name of component, this is the last name segment of the fully qualified name for the MDO component involved in the last operation, if applicable.

&ZMDOTYPE
Type of &ZMDOCOMP, if applicable.

&ZMDOTAG
Tag value of component involved in the last operation, if applicable.

Input Operations on an MDO

A number of NCL verbs allow input operations on an entire MDO. These are:

- &APPC RECEIVE
- &ASSIGN
- &CNMREAD
- &DECODE
- &ENCODE
- &FILE GET
- &INTREAD
- &LOGREAD
- &MSGREAD
- &PPI RECEIVE
- &PPOREAD
- &VARTABLE GET

When the MDO is targeted for input the MAP operand is allowed to define the mapping of the data object being accessed. The state of the MDO following any such input operation is determined by a number of factors that apply generally to all verbs. Together these considerations produce an MDO behavior which is predictable, as follows.

- If the verb return code indicates that the request was not satisfied, either due to some error, or because no data satisfied the particular request (perhaps due to timeout, or selection criteria or similar), the target MDO is deleted. Subsequent reference to the MDO or its components is invalid, and will produce a name check return code.
- If the verb return code indicates that the request was satisfied, then the target MDO always exists, even if it is null (or empty). If no map name was supplied on the input operation, and no default map name applies, then the MDO is unmapped. If a map name was supplied but either the map could not be found, was in error, or the data did not conform to the map definition, then the map check return code is set with a feedback indicating the nature of the error. If the map was not found, the data is present in the MDO which is unmapped. Otherwise, if no errors are encountered, the MDO will exist and is mapped according to the map name implied.

After performing a successful input operation on an MDO, the &ZMDORC system variable should always be checked to ensure that the outcome was good, and that the MDO is still mapped. Failure to do so can cause the NCL procedure to be terminated if a reference to an MDO component is made, the MDO is unmapped, and &CONTROL MDOCHK is in effect.

However, once the MDO is bound to the map without error, its contents are guaranteed valid by Mapping Services and there is usually no need to check the return codes for every access to MDO components, but they are available if required.

Output Operations from an MDO

A number of NCL verbs allow output operations from an entire MDO. These are:

- &APPC SEND
- &ASSIGN
- &CNMALERT
- &CNMSEND
- &DECODE
- &ENCODE
- &EVENT
- &FILE PUT
- &PPI SEND
- &VARTABLE PUT
- &WRITE

Read exceptions from an MDO are rare, and are confined to a name check if an undefined component is referenced, or a data check if the data does not conform to the mapping rules. However, in some instances where the component definitions of fixed fields overlap each other type checks on read are possible.

Using the &ASSIGN Verb

The &ASSIGN verb provides the only access to and from individual components within an MDO. For a complete description of the &ASSIGN verb, see the *Network Control Language Reference.*

Creating and Deleting MDOs

To create a mapped MDO using the &ASSIGN verb the following statement should be issued:

```
&ASSIGN MDO=mdo MAP=mapname [ DATA=data ]
```

or

```
&ASSIGN MDO=mdo MAP=mapname [ FROM VARS=vars...  ]
```

To copy an MDO:

```
&ASSIGN MDO=mdo FROM MDO=sourcemdo
```

To create an unmapped MDO:

```
&ASSIGN MDO=mdo DATA=data
```

To delete an MDO entirely:

```
&ASSIGN MDO=mdo
```

Assigning Data into an MDO

The &ASSIGN OPT=DATA option is the default and can be used to set MDO components from:

- Another MDO component
- One or more NCL variables
- User supplied data

When assigning into an MDO component of a simple type from NCL variables or constant data the input supplied must be in the valid *external form* for the component type or a type check results.

When assigning into an MDO component of a simple type from another MDO component the input selected must have a *local form* valid for the target component type. If the types are different, type conversion will take place where possible, otherwise a type check results.

When assigning into an MDO component that is a constructed type the data must be valid in its physical format according to the structuring rules for the target component, and each embedded component must be in its valid *local form*, otherwise a type check results.

The external form and local form of data for each type is described in the chapter titled *Using Mapping Services in NCL* in the *MODS Programming and Administration Guide*.

Assigning into/from a Single MDO Component

The following assign statements can be used to set the value of a single target MDO component, which can be the entire MDO, from NCL variables or constant data:

```
&ASSIGN MDO=a.b.c DATA=xxx

&ASSIGN MDO=a.b.c FROM VARS=vars...
```

In either case the input string must be valid external form for the component or a type check results.

When multiple NCL variables are specified as the source data they are concatenated together to form the input string. The exception to this is if the assignment is into an entire MDO mapped by $NCL, in which case a standard variable structure is built and maintains the variable boundaries.

The following assign statement can be used to get the value of a single target MDO component, which can be the entire MDO, into NCL variables:

```
&ASSIGN VARS=vars FROM MDO=a.b.c
```

In all cases the result will be valid external form for the component unless a type check or data check occurs.

When multiple NCL variables are specified as the target data they are segmented according to the maximum variable size (or specific segment sizes if supplied) from the entire output string. The exception to this is if the assignment is from an entire MDO mapped by $NCL, in which case the NCL variables are updated according to the variable boundaries within the MDO.

The following assign statement can be used to move the value of a single target MDO component to another MDO component:

```
&ASSIGN MDO=a.b.c FROM MDO=x.y.z
```

In this case the assignment takes place using the normal local form for the component (not the external form) and as usual unless the input is valid, a type check results.

Assigning into/from Multiple MDO Components Within a SEQUENCE or SET Type

When an MDO component is a structure defined with a type of SEQUENCE or SET it is possible to assign into, or from, some or all of the components that comprise the structure by name. This is a generic form of assign, and the possible options and syntax is as follows:

```
&ASSIGN MDO=a.b.*
      { GENERIC | ADD | REPLACE | UPDATE }
      DATA=data
```

or

```
&ASSIGN  MDO=a.b.*
        FROM
        VARS=vars*
      { GENERIC | ADD | REPLACE | UPDATE }
```

or

```
&ASSIGN MDO=a.b.*
      { GENERIC | ADD | REPLACE | UPDATE }
      { FROM | PRESENT_IN | DEFINED_IN }
      MDO=x.y.*
```

or

```
&ASSIGN VARS=vars*
       { GENERIC | ADD | REPLACE | UPDATE }
       { FROM | PRESENT_IN | DEFINED_IN }
       MDO=x.y.*
```

Note

The asterisk (*) must be in place of the last component name only.

In all cases:

- The multiple assignment proceeds as though a separate assignment was issued for each component selected.
- The PRESENT_IN and DEFINED_IN keywords apply only to a source MDO, not a target MDO.
- The GENERIC, ADD, REPLACE and UPDATE keywords affect target MDO components or NCL variables only.

For all options, when assigning data into an MDO, the process is driven by the components defined within the map for the target MDO. Each component defined within the parent structure is a target for an assign. If the DATA keyword is used as the source, all target components are subject to assignment of the same data value. If the VARS keyword is used as the source, each target component name is used to access a source NCL variable with a name constructed by appending the component name to the supplied VARS prefix. If the MDO keyword is used as the source, each target component name is used to access a source component with a name constructed by adding the component name as the last name segment in the generic source MDO name.

When selecting data for assignment from an MDO the FROM keyword is used to select only source components that have a non-null data value. The PRESENT_IN keyword is used to select from all source components that are present, that is, those that have a null or non-null value. For both FROM and PRESENT_IN options, any component that is defined but not present is deemed to be null. When the DEFINED_IN keyword is used all components defined in the target map for the generic name level indicated take part in the operation, regardless of whether or not any data exists, and components not defined are deemed to be null.

When assigning data from an MDO into NCL variables, the process is driven by the components defined in the source MDO. Target NCL variable names are constructed by appending selected component names to the supplied VARS prefix.

When the GENERIC keyword is used any existing target NCL variables or MDO components are first deleted, then each is assigned the value from the corresponding source component If no source data exists no assignment takes place.

When the ADD keyword is used, only those NCL variables not currently present, or MDO components defined but not currently present in the target structure, take part in the assignment process. That is, only new NCL variables or MDO components are added and no existing ones are affected.

When the REPLACE keyword is used, only those NCL variables that are present, or MDO components that are defined and are present in the target structure, take part in the assignment process. That is, no addition takes place. Only existing NCL variables that have new source data or existing MDO components that have new source data, are affected, but those that have no new source data are not affected.

When the UPDATE keyword is used, both addition and replacement takes place, but existing NCL variables or MDO components that have no new source data are unaffected.

Assigning into/from Components Within a SEQUENCE OF or SET OF Type

When an MDO component is a structure defined with a type of SEQUENCE OF or SET OF it is possible to assign into, or from, some or all of the components that comprise the structure by a generic index value. This is a form of assign using a varying range, and the syntax is as follows:

```
&ASSIGN { MDO=a.b.{*} | VARS=aaa*}
                  RANGE=(n,m)
                  { DATA=data
                  | FROM  { VARS=bbb* | MDO=x.y.{*} }
                  [ RANGE=(p,q) ] }
```

Note

The asterisk in braces ({*}) must replace a SET OF or SEQUENCE OF index only. It might appear once only anywhere within the MDO name referenced.

The target component names of the form $a.b.\{i\}$, where $i = n$ up to m, take part in the assignment from the corresponding source variable. The multiple assignments take place as though a separate assignment was issued for each item within the SET OF or SEQUENCE OF structure. The variable index can be the last part of the MDO name (as shown in the example above), or more name segments can follow (for example, MDO=*a.b.*{*}.*c*).

Querying MDO Components

Once an MDO is connected to a map, it is possible to query its structure (as present in the MDO), or its definition (as defined in the map). The syntax used is part of an &ASSIGN, where the results of the query must be placed into NCL variables.

```
&ASSIGN OPT={ NAMES
                  |TAGS
                  |TYPE
                  |LENGTH
                  |#ITEMS
                  |NAMEDVALUES
                  |VALIDVALUES  }
      VARS=vars...
      { PRESENT_IN | DEFINED_IN | MANDATORY }
      MDO=mdoName
```

When the PRESENT_IN keyword is specified the information is returned only for those components that are found to be present within the MDO. When the DEFINED_IN keyword is used, the information is returned for all those components defined within the connected map, regardless of their presence or absence in the MDO itself. If the MANDATORY keyword is used then only those *defined* components that are mandatory are selected.

The use of the various options and their meanings are described below:

OPT=NAMES (or OPT=NAME)

Applies to PRESENT_IN, DEFINED_IN and MANDATORY options and returns the component names associated with the target *mdoName* as follows:

- If only an MDO stem name is specified (for example, MDO=*abc*) then the name of the currently connected map is returned as the defined component name on this query, but only if the MDO exists.
- If the *mdoName* is a compound name (for example, MDO=*a.b.c*), the name of the last component in the name list is returned (that is, "*c*"), depending upon the PRESENT_IN, DEFINED_IN or MANDATORY option.
- If the *mdoName* is a compound generic name (for example, MDO=*a.b.c.*$*$), multiple names can be returned, where each name returned is a sub-component of the nominated component (for example, for MDO=*a.b.c.*$*$, all "*x*" where "*x*" is a component defined within "*c*"). ,This is useful in determining the names of all components that are either present in, or defined within, a given structure. It is also useful in determining which component is within a structure that is a CHOICE type. Note however, that for SEQUENCE OF and SET OF items it is possible to have null named components as the SEQUENCE or SET OF items are processed by index value only.

A compound variable indexed name (for example, MDO=*a.b*{*}.*c*, or MDO=*a.b.*{*}) is not supported on this query.

OPT=TAGS

Applies to PRESENT_IN, DEFINED_IN and MANDATORY options and returns the component tags used by Mapping Services associated with the target *mdoName*. Component selection is as for OPT=NAMES.

OPT=TYPE

Applies to PRESENT_IN, DEFINED_IN and MANDATORY options and returns the component type defined within the map and associated with the target *mdoName*. Component selection is as for OPT=NAMES.

OPT=LENGTH

Applies only when PRESENT_IN is specified and returns the local form data length within the MDO of the target component(s). Component selection is as for OPT=NAMES.

OPT=#ITEMS

Applies only when PRESENT_IN is specified, and returns the number of items within a nominated component as follows:

- If *mdoName* is a stem name (for example, MDO=*stem*), or a compound name (for example, MDO=*a.b.c*), then a count of 0 is returned if the component does not exist, otherwise it is 1.
- If *mdoName* is a compound generic name (for example, MDO=*a.b.c.**), a count of 0 is returned if the nominated component *a.b.c* is one of the following:

  a. Does not exist
  b. Exists but is empty
  c. Exists but is not constructed

  Otherwise it provides the number of components present within the structure *a.b.c*.
- If *mdoName* is a compound variable indexed name (for example, MDO=*a.b*{*}, or MDO=*a.b.*{*}), the number of components present in the SET OF or SEQUENCE OF structure is returned, or 0 if the structure does not exist or is empty. The variable index must be in the last name segment.

OPT=NAMEDVALUES

Applies to components that have named values associated with their type. These types are limited to BIT STRING, INTEGER, and ENUMERATED. Other types will return null results. No generic indexes or generic names are allowed on this option.

If DEFINED_IN is specified a list of the named values defined in the map for the specified component is returned.

PRESENT_IN is invalid for OPT=NAMEDVALUES.

OPT=VALIDVALUES

Applies to string types that can have their character set constrained to particular characters or strings. It only works in conjunction with the defined keyword. If a string type (for example, GraphicString) has been constrained to a particular set of characters or strings, then this option returns the valid characters or strings in the target variable. If there are no constraints then no values are returned on assignment. The &ZVARCNT system variable is set to indicate the number of target variables set by the assignment.

Example 1:

If there is a component defined as follows:

```
datax GraphicString ("ABCD" | "xyz" | "QQQ")
```

then &ASSIGN VARS=X* OPT=VALIDVALUES DEFINED MDO=... datax will return 3 variables set as follows:

```
&X1=ABCD
&X2=xyz
&X3=QQQ
```

Example 2:

If there is a component defined as follows:

```
datax GraphicString (FROM ( "A"c | "C" | "Y"C | "X" ))
```

then &ASSIGN VARS=X* OPT=VALIDVALUES DEFINED MDO=... datax will return 4 variables set as follows:

&X1=A
&X2=C
&X3=Y
&X4=X

NCL Reference, Type Checking, and Data Behavior

When referencing an MDO in an NCL procedure, Mapping Services validates that the named component is defined (according to the name hierarchy supplied), and that the data within the component is valid, according to its underlying ASN.1 type. Each ASN.1 type can contain only certain valid values. Mapping Services checks the data value when retrieving data from, or assigning data into, an MDO. An operation attempting to retrieve or assign invalid data is rejected by Mapping Services with a feedback indicating *type check*.

In order to perform type checking Mapping Services first determines the base ASN.1 type of the component. Where a component is of a user defined type, the base ASN.1 type of the user defined type is inherited by the component. It is possible to have a number of levels of indirection between a user defined type and its base ASN.1 type.

The valid *NCL values* allowed for each of the base ASN.1 types is termed the *external form*. In addition to the set of valid values for each type, a specific component can be further constrained in what values are acceptable. Such *constraints* can be the result of either ASN.1 definitions or compiler directives. Finally, when data representing a valid NCL value is accepted for a component update, it is subject to a transformation from external form to *local form*, which is the MDO internal representation of data. This process can carry with it further constraints.

The valid external form values, and the behavior of data managed by Mapping Services, is described for each type in the following sections.

Note

In the following descriptions, all string types that are defined as fixed (using the --<FIX(*n*)>-- directive) are subject to padding and truncation, without any indication in the return codes.

Any types constrained by the SIZE parameter are not subject to padding or truncation. The data supplied must be within the SIZE constraints specified, or a *type check* results.

The BOOLEAN Type

Use

The BOOLEAN type is used to represent a value of *true* or *false* only.

External Form - Input

The local character strings TRUE and FALSE (not case sensitive) are accepted, while the digit 0 is interpreted as false, and the digit 1 is true.

External Form - Output

The digit 0 (false) or 1 (true) is always returned.

Local Form and Behavior

Internally, Mapping Services stores a value of X'00' for false, and X'01' for true (and accepts any value other than X'00' as true).

For an input operation, where the component is variable length, its length is always set to 1. Where the component length is fixed and is greater than 1, the value occupies the first byte only (that is, it is left aligned) and the remainder of the component's data is set to zeros.

For an output operation, where the component is located and has a length greater than 1, only the first byte is inspected as the value.

The INTEGER Type

Use

The INTEGER type is used to contain any positive or negative whole numbers in the range -2,147,483,648 to 2,147,483,647 (that is, it is a signed 32 bit number).

External Form - Input

Valid input consists of a string of up to 15 digits optionally preceded by a plus sign (+) or minus sign (-) providing the sign (positive or negative) of the value (positive if omitted). All other characters must be valid digits (that is, 0..9). Alternatively, if the map definition included named values for this component, the symbolic name of the named value can be supplied as external form input.

External Form - Output

Output consists of a string of one or more local characters. If the integer value is negative the first character is a minus sign (-), otherwise the sign is omitted. All other characters are numeric characters. Leading zeroes are stripped.

Local Form and Behavior

Internally, Mapping Services can store integers in one of three formats:

binary

can be up to 4 bytes in length. If the length is not fixed then the value is kept in the smallest number of bytes possible. If the length is fixed then the value is right-aligned and sign extended to the left.

packed

can be up to 8 bytes in length. The integer value is converted to the packed decimal equivalent. If the length is not fixed then the value is kept in the smallest number of bytes possible. If the length is fixed then the value is right-aligned and zero padded to the left.

zoned

can be up to 15 bytes in length. The integer value is converted to the zoned decimal equivalent. If the length is not fixed then the value is kept in the smallest number of bytes possible. If the length is fixed then the value is right aligned and zero padded to the left.

For any format, if a value exceeds that which can be stored without loss of significance a type check results. If a named value is input then the map definition is used to determine the actual integer value.

The BIT STRING Type

Use

The BIT STRING type is used to contain any data where individual bit values might have meaning. Mapping Services supports two types of BIT STRING access, standard and boolean.

Standard BIT STRING Access

Use

Standard BIT STRING access deals with the string as a whole, allowing manipulation of the entire component through a single operation, as for most other types.

External Form - Input

Valid external form can be a string of one or more digits, each a 0 or 1. However, where named values are defined for the BIT STRING type, a list of named values, each separated by a plus sign (+) or a minus sign (-), is an acceptable alternative. A named value preceded by a plus sign indicates that the named bit value should be set to true (the bit is set to 1), and a minus sign indicates that the value should be set to false (the bit is set to 0).

External Form - Output

The output format depends upon whether or not named values are defined for the BIT STRING type. Where no named values are defined the output consists of a string of zero or more (always a multiple of 8) digits, each a 0 or 1. Where one or more named values do exist the output is a character string comprised of every named value corresponding to a set bit preceded by a plus sign meaning the value is true (the named bit is 1). The names of bits which are not set are not returned.

Local Form and Behavior

When a string if 0's and 1's are supplied as input, each digit in the input sequence is treated (left to right) as the value of the corresponding bit in that position of the local form data. If the number of bits supplied is not a multiple of 8, then trailing bits are set to zero and padded to a byte boundary. If the component has a fixed length exceeding that of the input string the value is left aligned, and all unreferenced bytes are set to X'00'. If the component cannot contain the number of input bytes supplied, a data check results.

When a list of named values, each preceded by a plus sign or a minus sign, is supplied as input only the named bits take part in the operation. Each named bit preceded by a plus sign is set to 1 (true), while each bit preceded by a minus sign is set to 0 (false). All other bits in the BIT STRING are unaffected by the input operation.

When fetching the value of a BIT STRING a named value list is always returned if any named values are defined for the type, else a string of 0's and 1's is returned corresponding to the BIT STRING values. Note that when named values are defined all other bits in the BIT STRING are ignored on output regardless of their value.

Boolean BIT STRING Access

Use

Boolean BIT STRING access deals with individual bit level access and operates only through named values. This access is recommended because program access to bits is only via their symbolic named values, thus removing from NCL the need to know relative bit positions.

For Boolean BIT STRING access to be invoked, the named value of a bit is provided by NCL as an additional name segment after the BIT STRING component name. Since the BIT STRING type is primitive, the additional name in the name hierarchy is understood to be a named value, and is treated as a BOOLEAN type. No matter where the named value is in the BIT STRING the value of the bit is always 0 or 1, as for a BOOLEAN type.

External Form - Input

The local character strings TRUE and FALSE (not case sensitive) are accepted, while the digit 0 is interpreted as false, and the digit 1 is true. Null is a type check in this case.

External Form - Output

The digit 0 (false) or 1 (true) is always returned.

Local Form and Behavior

The component name plus the named value is treated as a reference to a specific bit (the bit position within the component being defined by the named value), and that bit is set to 0 or 1 depending upon the input. No other bits in the BIT STRING component are affected. If the component is extended to accommodate the input then all other bits are set to 0.

The OCTET STRING Type

Use

The OCTET STRING type is used to contain any data where no formatting is required.

External Form - Input

Any data.

External Form - Output

Data is returned unchanged.

Local Form and Behavior

Data is stored *as is*. If the component has a fixed length exceeding that of the input string, the data us left aligned, and all unreferenced bytes are set to X'00'. If the component cannot contain the number of input bytes supplied, a data check results.

The HEX STRING Type

Use

The HEX STRING type is a Mapping Services extension to ASN.1, but is processed as a base ASN.1 type. It is identical in all respects to the ASN.1 OCTET STRING type except for its external form representation.

External Form - Input

Valid input consists of a string of one or more local characters, each selected from the set 0123456789ABCDEF. Each pair of hexadecimal characters represents a single byte value. If an odd number of characters is supplied the string is treated as though padded on the left with a single zero.

External Form - Output

Data is returned in hexadecimal characters, as for input. An even number of characters is always returned.

Local Form and Behavior

Each two hexadecimal characters of input represents the actual data to be stored in a single byte. Otherwise behavior is as for OCTET STRING.

The NULL Type

Use

The NULL type is used where data in a component either must be null (that is, empty), or not accessible.

External Form - Input

The only valid input is a null value.

External Form - Output

A null value is always returned.

Local Form and Behavior

The component can be created by an input operation, but no contents are modified. If it already exists no data is modified.

The OBJECT IDENTIFIER Type

Use

The OBJECT IDENTIFIER type is used to contain object identifier values that uniquely identify registered objects.

External Form - Input

Any sequence of characters (from the set 0123456789.) that does not begin or end with a period (.), contains no consecutive periods, but contains at least one period. Each sequence of decimal digits punctuated by a period represents a sub-identifier in the series of sub-identifiers that comprise an object identifier value.

External Form - Output

As for input.

Local Form and Behavior

The data format is as supplied for input, however truncation is not allowed. If the component is fixed length then it must be able to contain the input string, and if necessary will be padded with blanks, otherwise a data check results.

The ObjectDescriptor Type

Use

The ObjectDescriptor type is used to contain object descriptions for registered objects.

External Form - Input

As supplied.

External Form - Output

As supplied.

Local Form and Behavior

The data format is as supplied for input. Normal string padding and truncation rules apply.

The REAL Type

Use

The REAL type is used to contain floating point, or scientific notation, numbers in the range 10-70 to 1070.

External Form - Input

Format allowed is:

+ or -	(optional, plus or minus sign, followed by)
nnnnnn	(optional, any number of digits, followed by)
period (.)	(optional, decimal place, followed by)
mmmmmm	(optional, any number of digits, followed by)
E*sxx*	(optional, signed exponent power of 10, range -99 to 99)

such that either, or both, *nnnnnn* and *mmmmmm* are present, and the resulting REAL number is within the allowable range.

Valid examples:

```
1457892345509676544283940 4
```

```
-123.567
```

```
.555
```

```
.0023E-23
```

```
3.142776589E+66
```

External Form - Output

Normalized decimal real number:

+ or -	(plus or minus sign of the value, followed by)
.	(decimal place indicator, followed by)
nnnnnn	(15 significant fraction digits, followed by)
E*sxx*	(signed exponent power of 10)

Examples:

```
+.314277658900000E-10

-.123456789000000E+52
```

Local Form and Behavior

For IBM S/370 machines, local form is a 64-bit *long floating point* value, and the component must be at least 8 bytes in length. Truncation is not allowed, but if the component has a fixed length greater than 8 bytes the value is left aligned and padded on the right with zero bytes.

The ENUMERATED Type

Use

The ENUMERATED type is used to constrain a component to a defined set of values. Each defined value is *named* using a name identifier similar to a component name. Associated with each name is a unique integer value (which can be signed), for example:

```
Color ::= ENUMERATED { red(0),blue(1),yellow(2),
            green(3),black(7) }
```

External Form - Input

The external form must be one of the names listed in the ENUMERATED type. The enumerated values are not allowed (that is, `red` is valid, `0` is not).

External Form - Output

Same as input.

Local Form and Behavior

Internally, the ENUMERATED value is kept in the same manner, and is subject to the same local form constraints, as an INTEGER of the binary local form.

The NumericString Type

Use

The NumericString is a subset of GeneralString which comprises:

0 to 9	(numeric characters)
( )	(the space, or blank character)

External Form - Input

Any string of valid characters as described above.

External Form - Output

Same as input.

Local Form and Behavior

On input data is stored as supplied. Normal string padding and truncation rules apply.

The PrintableString Type

Use

The PrintableString is a subset of GeneralString which comprises:

a to z	(lowercase alphabetic characters)
A to Z	(uppercase alphabetic characters)
0 to 9	(numeric characters)
( )	(the space, or blank character)
’ ( ) + , - . / : = ?	(special characters)

External Form - Input

Any string of valid characters as described above.

External Form - Output

Same as input.

Local Form and Behavior

On input data is stored as supplied. Normal string padding and truncation rules apply.

The TeletexString Type

Use
None in NCL.

External Form - Input
As supplied.

External Form - Output
As supplied.

Local Form and Behavior
As for OCTET STRING.

The VideotexString Type

Use
None in NCL.

External Form - Input
As supplied.

External Form - Output
As supplied.

Local Form and Behavior
As for OCTET STRING.

The IA5String Type

Use
Transparent general character set. (VisibleString plus control characters).

External Form - Input
As supplied.

External Form - Output
As supplied.

Local Form and Behavior
On input data is stored as supplied. Normal string padding and truncation rules apply.

The UTCTime Type

Use

Date and time, as Universal Coordinated Time (year without century numbers), in the format:

`YYMMDDHHMM[`*`SS`*`]Z`
: GMT date and time (to minutes or seconds); Z indicates GMT time

`YYMMDDHHMM[`*`SS`*`]`*`s`*`HHMM`
: local date and time (to minutes or seconds); with signed zone offset from GMT time (*s* = + or -)

External Form - Input

Any valid input as shown above.

External Form - Output

Any valid data as shown above.

Local Form and Behavior

On input data is stored as supplied. If the component has a fixed length then a short input string will be padded with blanks. Truncation is not allowed.

The GeneralizedTime Type

Use

Date and time, as GeneralizedTime (year includes century numbers), in the format:

`YYYYMMDDHH[`*`MM`*`[`*`SS`*`]][.`*`f`*`]Z`
: GMT date and time (to hours, minutes or seconds); with optional fractional time units to any significance (hours, minutes or seconds); Z indicates GMT time

`YYYYMMDDHH[`*`MM`*`[`*`SS`*`]][.`*`f`*`][`*`sHHMM`*`]`
: local date and time (to hours, minutes or seconds); with optional fractional time units to any significance (hours, minutes or seconds); with signed zone offset from GMT time (*s* = + or -)

External Form - Input

Any valid input as shown above.

External Form - Output

Any valid data as shown above.

Local Form and Behavior

On input data is stored as supplied. Normal string padding and truncation rules apply.

The GraphicString Type

Use

Transparent character set of graphic characters only.

External Form - Input

As supplied.

External Form - Output

As supplied.

Local Form and Behavior

On input data is stored as supplied. Normal string padding and truncation rules apply.

The VisibleString Type

Use

Transparent character set of graphic characters only.

External Form - Input

As supplied.

External Form - Output

As supplied.

Local Form

On input data is stored as supplied. Normal string padding and truncation rules apply.

The GeneralString Type

Use:

Transparent character set of both graphic and control characters.

External Form - Input

As supplied.

External Form - Output

As supplied.

Local Form

On input data is stored as supplied. Normal string padding and truncation rules apply.

Type Conversion for MDO Assignment

Normally when assigning data from one MDO component to another the type of each component is identical so that data can be moved unchanged. However, if the target component is not of the same type as the source component type conversion is automatically performed where possible. Hence the result of the following assignment depends on the type of *a.b.c* and *x.y.z*:

```
&ASSIGN MDO=a.b.c FROM MDO=x.y.z
```

When assigning into an MDO from external data, or from NCL variables the data type of the input can be nominated for the MDO assignment. If the MDO component is not of this type then type conversion is performed, for example:

```
&ASSIGN MDO=a.b.c TYPE=INTEGER FROM DATA=123

&ASSIGN MDO=a.b.c TYPE=HEXSTRING FROM VARS=HEXDATA
```

Similarly, when assigning data from an MDO into NCL variables, the data type of the output can be nominated for the MDO assignment, for example:

```
&ASSIGN VARS=ABC FROM MDO=a.b.c TYPE=BITSTRING
```

Valid type operands are shown in the following table:

BOOLEAN	or BOOL	
INTEGER	or INT	
BITSTRING	or BIT	
OCTETSTRING	or OCTET	
NULL		
OBJECTIDENTIFIER	or OBJECTID	or OID
OBJECTDESCRIPTOR	or OBJECTDESC	or ODESC
REAL		
HEXSTRING	or HEX	
NUMERICSTRING	or NUMERICSTR	or NUMSTR
PRINTABLESTRING	or PRINTABLESTR	or PRTSTR
IA5STRING	or IA5STR	
UTCTIME	or UTC	
GENERALIZEDTIME	or GTIME	
GRAPHICSTRING	or GRAPHICSTR	or GRAPHSTR
VISIBLESTRING	or VISIBLESTR	or VISSTR
GENERALSTRING	or GENERALSTR	or GSTR

Type conversion is attempted regardless of the source and target types. Thus, depending on the actual value of the source data and target type, some assignments might produce valid results, while others will produce a type check.

The ASN.1 types can be classified into 3 groups:

- Graphic-oriented types:
  OBJECT IDENTIFIER, OBJECT DESCRIPTOR, UTCTime, GeneralizedTime, NumericString, PrintableString, TelexString, VideotexString, IA5String,GraphicString, VisibleString and GeneralString.
- Numeric-oriented types:
  BOOLEAN, INTEGER, BIT STRING, REAL and ENUMERATED.
- Transparent types:
  OCTET STRING, HEX STRING, SEQUENCE (OF), SET (OF), and ANY.

The following table shows the general rule for type conversion. It lists the formats of the source data value used for assignment into the target type.

SOURCE\\TARGET	Graphic-oriented	Numeric-oriented	Transparent
Graphic-oriented	External form	External form	Local form
Numeric-oriented	External form	Local form	Local form
Transparent	Local form	Local form	Local form

See the section, *NCL Reference, Type Checking, and Data Behavior*, on page 9-14, for details on the acceptable external and local form values for each type.

Graphic-oriented Source Type

Where the target is one of the graphic- or numeric- oriented types, the external form output of the source data is assigned to the target as though it was external input. The following is an exception:

- Conversion between UTCTime and GeneralizedTime results in the insertion/stripping of the century value (insert 19 if *yy* greater than 50, otherwise insert 20)

Where the target is one of the transparent types, the local form of the source data is assigned to the target unchanged; that is, the local form of the source data becomes the local form of the target. However, the external outputs of the source and target can differ depending on their respective types. In either case, a type check resultS if the data is invalid for the source or target types. For example:

```
&ASSIGN MDO=a.b.utc TYPE=GTIME DATA=19921012145000+1000
                    -* target type is UTCTime
&ASSIGN VARS=RESULT FROM MDO=a.b.utc
                    -* returns &RESULT = 921012145000+1000

&ASSIGN MDO=a.b.num DATA=0123
                    -* source type is NumericString
&ASSIGN VARS=RESULT FROM MDO=a.b.num TYPE=REAL
                    -* output type is REAL
                    -* returns &RESULT = +.1230000000000000E+03

&ASSIGN MDO=a.b.graph DATA=ABCD
                    -* source type is GraphicString
&ASSIGN MDO=a.b.hex FROM MDO=a.b.graph
                    -* target type is HEX STRING
&ASSIGN VARS=RESULT FROM MDO=a.b.hex
                    -* returns &RESULT = C1C2C3C4
```

Numeric-oriented Source Types

Where the target is one of the numeric-oriented or transparent types, the local form of the source data is converted to the local form of the target. The source and target local form values are equal in most cases, although their external form output can differ depending on their respective types. The following, however are exceptions:

- If the source type is REAL and the target type is either BOOLEAN, INTEGER or ENUMERATED, then floating point integer value conversion (with rounding) is performed on the local form of the source data. Similarly, integer to floating point conversion is performed if the process is reversed.
- Conversion of local form source data to a BOOLEAN type target, results in X'00' (FALSE) if the source is non-zero. Otherwise, the target local form is converted to X'01' (TRUE).

Where the target is one of the graphic-oriented types, the external form output of the source data is assigned to the target as though it was external input.

In either case a type check results if the data is invalid for the source or target types.

For example:

```
&ASSIGN MDO=a.b.int TYPE=REAL DATA=99.99
                    -* target type is INTEGER
&ASSIGN VARS=RESULT FROM MDO=a.b.int
                    -* returns &RESULT = 100 (value rounded)

&ASSIGN MDO=a.b.bool DATA=TRUE
                    -* source type is BOOLEAN
&ASSIGN VARS=RESULT FROM MDO=a.b.bool TYPE=HEX
                    -* output type is HEX STRING
                    -* returns &RESULT = 01 (that is, TRUE)

&ASSIGN MDO=a.b.bit DATA=11101
                    -* source type is BIT STRING
&ASSIGN MDO=a.b.num FROM a.b.bit
                    -* target type is NumericString
&ASSIGN VARS=RESULT FROM MDO=a.b.num
                    -* returns &RESULT = 11101000
```

Transparent Source Types

Regardless of the target type, the local form of the source data is assigned to the target unchanged, that is, the local form of the source data becomes the local form of the target. The following is an exception:

- Conversion of local form source data to a BOOLEAN typed target results in local form X'00' (FALSE) if the source is non-zero. Otherwise, the target local form is converted to X'01' (TRUE).

Note that the external outputs of the source and target values can differ, depending on their respective types.

A type check results if the data is invalid for the source or target types.

For example:

```
&ASSIGN MDO=a.b.prt TYPE=OCTET DATA=AB.C
                    -* target type is PrintableString
&ASSIGN VARS=RESULT FROM MDO=a.b.prt
                    -* returns &RESULT = AB.C

&ASSIGN MDO=a.b.hex DATA=FF
                    -* source type is HEX STRING
&ASSIGN VARS=RESULT FROM MDO=a.b.hex TYPE=INT
                    -* target type is INTEGER
                    -* returns &RESULT = -1

&ASSIGN MDO=a.b.any DATA=DEFG
                    -* source type is ANY
&ASSIGN MDO=a.b.hex FROM MDO=a.b.any
                    -* target type is HEX STRING
&ASSIGN VARS=RESULT FROM MDO=a.b.hex
                    -* returns &RESULT = C4C5C6C7
```

10

System Level Procedures

This chapter discusses the following topics:

- System Level Procedures
- The Message Profile Concept
- Using LOGPROC to Intercept Log Messages
- Intercepting Solicited and Unsolicited VTAM Messages (PPOPROC Procedures)
- Intercepting OCS Messages (MSGPROC Procedures)
- User ID Considerations for System Level Procedures

System Level Procedures

NCL can be used to write *private* programs that are executed by command from your OCS windows, or implicitly from options such as the User Services panel.

Another class of NCL process called *system level procedures* executes exclusively in the NCL processing regions of certain background virtual users. These system level procedures have access to the special information flows that occur within a SOLVE system, namely:

- The Management Services log
- VTAM messages
- EASINET terminal control data
- OCS window traffic

The Management Services Log

The stream of output messages to the Management Services log provides a serial record of all activity in the SOLVE system. Therefore, it is an ideal source of real-time information about SOLVE events. While the log is accessible for on-line review by real operators, there is a special background *virtual user environment* called LOGPROC that directly accesses every message written to the Management Services log through special NCL verbs:

- The SYSPARMS LOGPROC command nominates the name of a procedure that is to act as the LOGPROC procedure. As such, it can execute these special verbs to see and modify messages destined for the Management Services log.
- LOGPROC is a system level procedure (one per SOLVE system) that can act as a central intelligence point to monitor events being reported to the log. As such, it can observe and react to unusual or critical conditions, and provide a platform for automatic recovery action.

VTAM Messages

The VTAM *primary program operator* (PPO) interface allows Management Services to receive unsolicited messages from VTAM about network events that occur for reasons other than from operator commands. For example, if an NCP fails, VTAM will generate unsolicited messages notifying the PPO application of the occurrence.

The PPO interface can also be fed with commands and responses resulting from operator action at the system console and from OCS windows within the SOLVE system.

The stream of PPO messages to Management Services represents a serial record of all unexpected events within the VTAM network. As such it is an ideal source of information about real time network failures.

While PPO messages can be routed automatically to OCS and system console operators for their attention, there is also a virtual user environment within Management Services called PPOPROC that directly accesses every message (or a chosen subset of messages) received from VTAM. This PPO message flow is accessed only through special NCL verbs:

- The SYSPARMS PPOPROC command nominates the name of a procedure that is to act as the *PPOPROC procedure*, and as such can execute these special verbs to see and modify PPO messages received from VTAM.
- PPOPROC is a system level procedure (one per SOLVE system) that acts as a central intelligence point to monitor events being reported by VTAM. As such, it can observe and react to unusual or critical conditions and provide a platform for the collection of additional information about the event and for the initiation of automatic recovery action.

EASINET Terminal Control

The EASINET feature lets you place all idle network terminals under Management Services control whenever they are not natively logged on to another network application.

The manner in which terminals are handled by the EASINET feature is determined by the logic of the EASINET procedure. The EASINET procedure, identified by the SYSPARMS EASINET command, is a system level procedure and is executed on behalf of every terminal that logs on to Management Services.

OCS Window Traffic Handling

Each user operating one or two OCS windows is entitled (depending on their user ID profile) to nominate a system level procedure to help them filter and monitor messages sent to those windows.

These procedures are called MSGPROCs. The name of the procedure executed for a user to act as the MSGPROC for each user OCS window is defined as part of their user profile:

- A MSGPROC procedure has access to the serial stream of messages sent to the OCS window for which it is executing and can receive, modify, delete, or react to any message or message group sent to the window.
- MSGPROC procedures can therefore act as *preprocessors* for messages that an operator would otherwise have to monitor.

The Message Profile Concept

System level procedures access the messages flowing on their particular stream by issuing specialized NCL verb read requests that are unique to each type of system level procedure. For example, PPOPROC uses &PPOREAD to obtain the next PPO message, and LOGPROC uses &LOGREAD to obtain the next log message.

The processing that each system level procedure undertakes depends on analyzing the messages that are supplied to it. To help analyze the message received and decide the specific processing required, the concept of *message profiling* is used.

A message that satisfies a system level procedure read request has a profile containing the attributes associated with that message. For example, a MSGPROC message profile includes not just message text but other information about color, highlight options, or message origins that the MSGPROC procedure might want to know or change or base some decision upon.

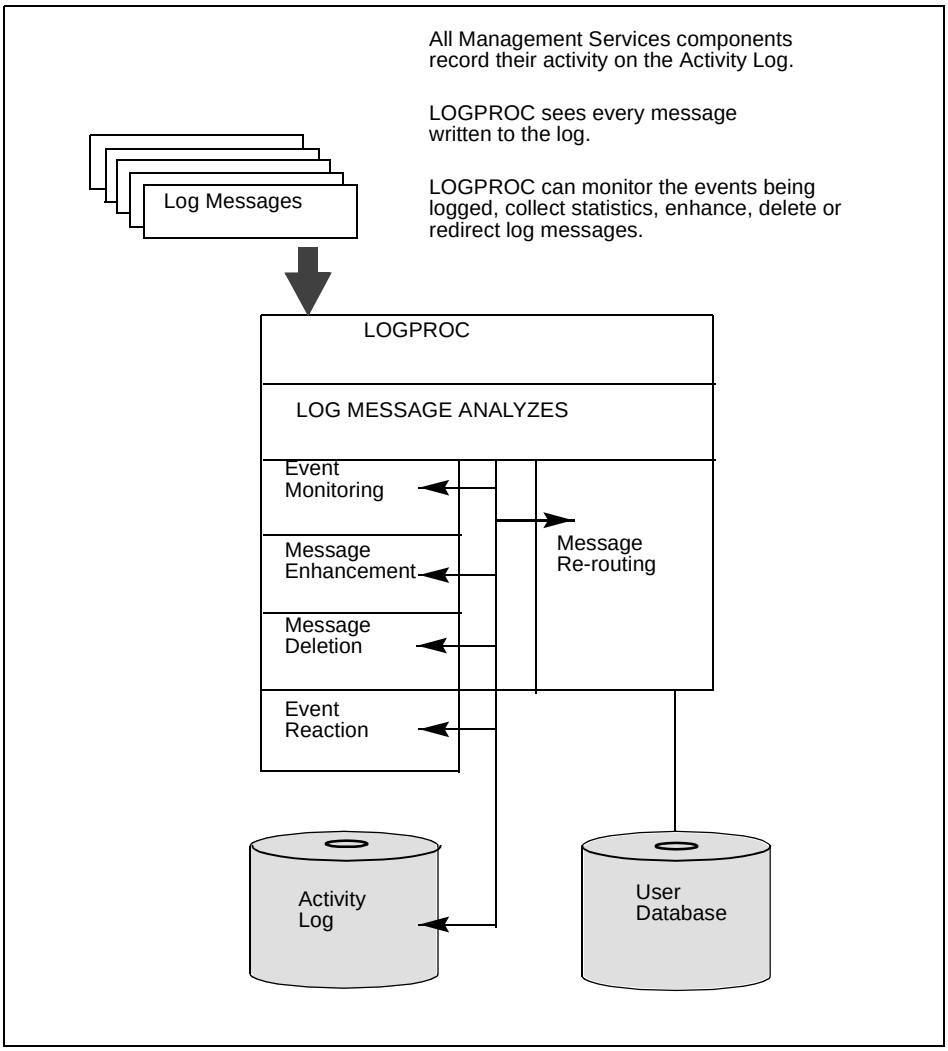
On completion of the read statement, all applicable message attributes are made available to the system level procedure as a series of reserved system variables known as *message profile variables*. An examination of the message profile variables is often sufficient to make a decision on whether to perform further analysis of the message, or to show that the message is of no interest to the procedure.

The message profile concept is described in Appendix D, *System Level Procedures: Message Profiles*, in this manual. This appendix details all message profile variables and the subsets of those variables that apply on completing different NCL verbs processing.

Using LOGPROC to Intercept Log Messages

Using LOGPROC lets you monitor system activity indirectly. An example of such activity is monitoring the completion of FTS file transmissions, or automatically generating and submitting JCL for a jobstream to process the data received. Figure 10-1 shows a schematic of LOGPROC operation.

Figure 10-1. LOGPROC Monitors all Activity Recorded on the Log



LOGPROC can also categorize the data written to system logs and write them to one or more UDBs for use when browsing on-line, or similar. A sample LOGPROC and its associated full-screen panels for performing on-line browsing of the log database is supplied with your SOLVE system.

Designing LOGPROC Procedures

A LOGPROC procedure should be written as a closed loop. When invoked, it processes until the first &LOGREAD statement is detected and then suspends processing until a LOG message arrives.

As each message arrives, the procedure logic should branch to an appropriate processing point for that message. After processing the message the procedure should then return to the initial &LOGREAD where processing is suspended once again.

If the logic of the LOGPROC procedure allows it to terminate, an error message is issued to monitor-status terminals and normal LOG processing resumes.

LOGPROC should be designed to avoid heavy processing of any particular events. If an event is logged that requires specific action, consider using started processes to handle the condition independently of the main LOGPROC process.

Note

If LOGPROC ends abnormally, an NRD message is dispatched to all monitor-status OCS users. It is unusual for LOGPROC to be designed to end, and using NRDs is likely to be more effective for warning Management Services operators that the process has terminated.

Messages from LOGPROC

Messages written from LOGPROC, including those issued by &WRITE statements or resulting from commands executed from within LOGPROC (unless &INTCMD is used), are displayed at all monitor-status terminals, prefixed with an L (the L replaces the normal M prefixed to monitor messages.)

Messages generated from within LOGPROC go directly to the log and are not presented to LOGPROC for processing. This prevents recursive looping.

LOGPROC Statements

Special processing verbs are provided for LOGPROC use:

- &LOGREAD retrieves the next log message for processing
- &LOGDEL deletes a log message
- &LOGCONT processes the log message normally
- &LOGREPL replaces log message text

How to use these verbs is described in the *Network Control Language Reference* manual.

LOGPROC can take advantage of the dependent processing environment facilities (using &INTCMD statements) to invoke independent processes (using START) to process particular conditions or events.

Testing LOGPROC

The SYSPARMS LOGPROC operand is used to invoke LOGPROC, and can also be used to flush it. This allows a new copy of the LOGPROC procedure to be invoked from a subsequent SYSPARMS command.

Note

The base LOGPROC procedure never uses a preloaded or retained copy of the procedure. Therefore it is unnecessary to use the SYSPARMS UNLOAD command. Other procedures invoked from within LOGPROC observe the normal preload conventions.

Intercepting Solicited and Unsolicited VTAM Messages (PPOPROC Procedures)

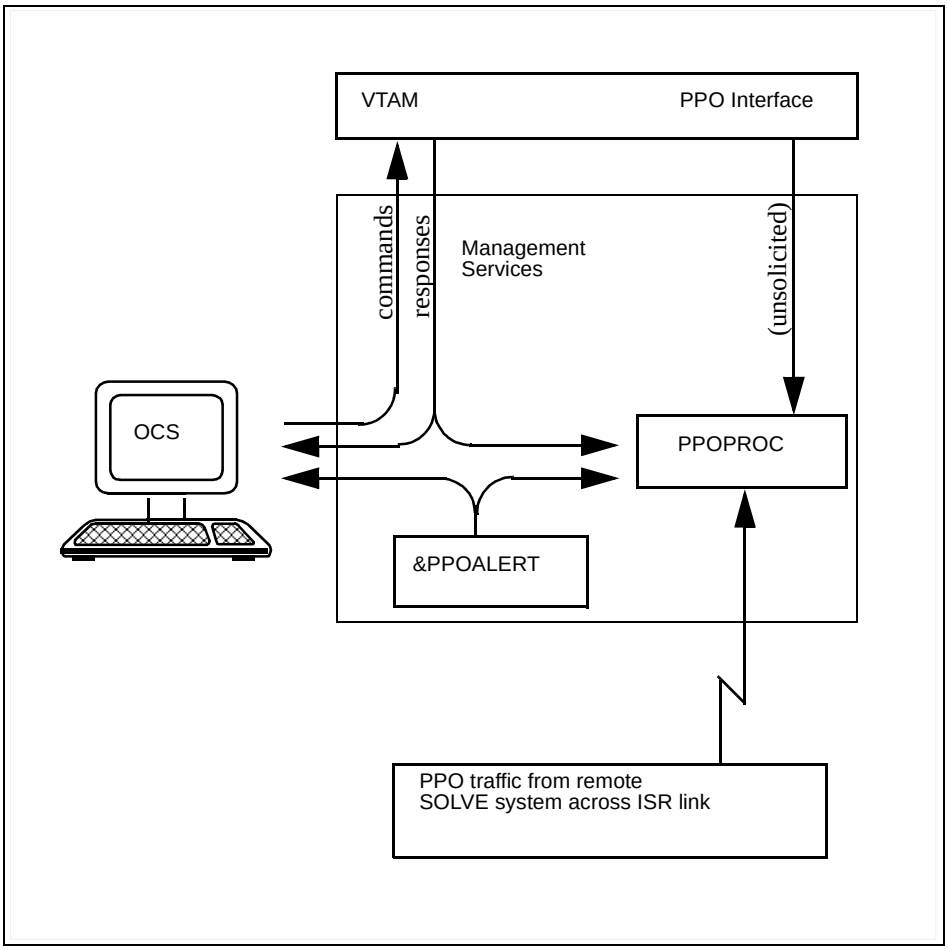
VTAM's primary program operator interface (PPO) can be defined so that Management Services receives both solicited and unsolicited VTAM messages. Management Services provides an NCL interface to the VTAM PPO facility called PPOPROC. PPOPROC can use this information to monitor status changes of VTAM resources and is used by features such as NET/STAT for this purpose. By being aware of status changes, PPOPROC can be the primary source of decisions controlling automated reaction to events that occur within the network.

PPOPROC has four possible sources of information:

- The VTAM PPO interface which delivers both unsolicited VTAM-generated messages and optional, solicited messages from VTAM commands issued at system consoles. This depends on the VTAM PPOLOG initialization parameter.
- Copies of commands and command responses resulting from operator activity within Management Services.
- Messages delivered to PPOPROC from &PPOALERT statements issued by another NCL process within Management Services. This facility generates test messages and delivers them to PPOPROC for development and testing. It also lets you generate a new PPO message with more or less information than the original.
- PPOPROC related messages of the three preceding types routed from remote SOLVE systems across the Inter-System Routing (ISR) facility.

Figure 10-2 summarizes the sources of messages received by PPOPROC.

Figure 10-2. PPOPROC Message Sources



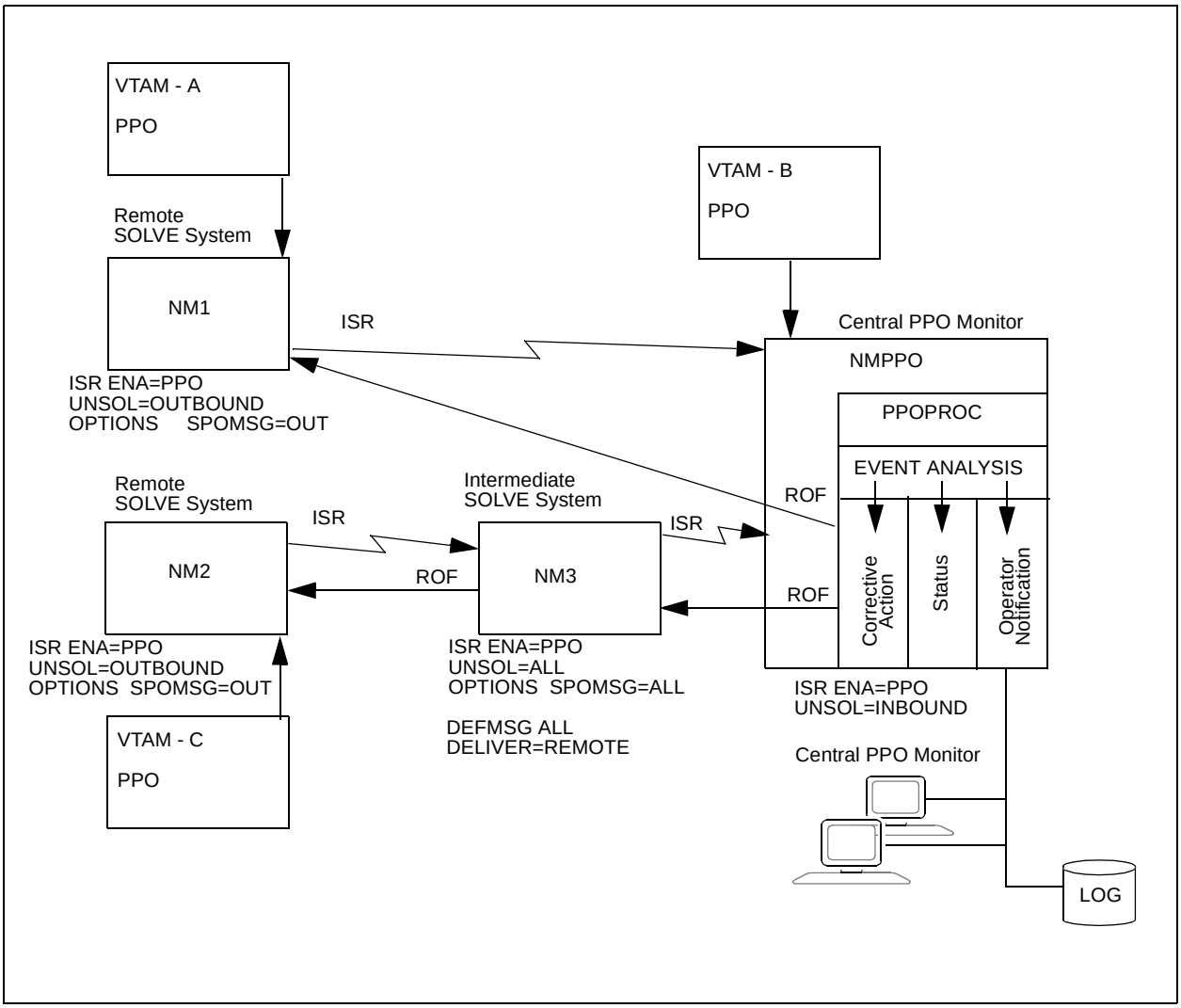
PPOPROC messages from these different sources can be used to report network resource status changes to a central point, regardless of whether the change is unexpected (in which case VTAM reports the event as an unsolicited message), or in response to an operator command.

Centralization can be achieved by routing all relevant PPO messages to a central SOLVE system. Several ISR options are used to facilitate this.

The central Management Services for monitoring PPO messages should be set up with the ISR ENABLE=PPO UNSOLICIT=INBOUND command. This allows it to receive PPO messages from remote SOLVE systems. The remote systems should specify ISR ENABLE=PPO UNSOL=OUTBOUND which will enable delivery of messages to the central PPO processing link.

Figure 10-3 shows an example of centralized PPO status monitoring for a multi-domain network.

Figure 10-3. PPO Status Monitoring in Multi-Domain Networks



In this figure, links have been established between NM1 and NMPPO, NM3 and NMPPO, and NM2 and NM3. NM3 acts as an intermediary between NM2 and NMPPO. All SPO messages occurring as a result of operator commands are routed to NMPPO and can be processed by its PPO procedure. PPO messages from VTAM-A defined for remote delivery by NM1 (either by the DEFMSG DELIVERY options or a PPOPROC override), are routed to NMPPO.

The DEFMSG command should be used to limit message flows across ISR to only those of interest to the PPO monitor. Similarly, PPO messages from VTAM-C are delivered to NM3 from NM2. NM3 can then pass them on to NMPPO. PPOPROC in NMPPO analyzes all VTAM messages and can take appropriate action, such as logging, or routing recovery commands to the remote systems.

Filtering Messages Seen by PPOPROC

PPOPROC is not necessarily interested in every message that could be sent to it. To limit the number of messages PPOPROC has to process, Management Services uses the *message definition table* (DEFMSG table) which specifies VTAM message numbers that PPOPROC wants to see.

Note

Very high message rates can occur at the PPO interface. Therefore the more you can decrease the number of messages delivered by the DEFMSG filter, the more efficiently your system will perform.

Modifying the Message Definition Table: The DEFMSG Command

The DEFMSG command lets you control message delivery for solicited and unsolicited VTAM messages. Three delivery destinations can be specified for each message:

- PPOPROC
- LOCAL (that is, to OCS operators)
- REMOTE (that is, systems linked by ISR)

In addition, *EDS events* can be defined in the DEFMSG table for VTAM messages.

A default class of messages has been pre-defined to the message table and initially set for delivery to all destinations. The delivery options for the default class messages can be altered using the DEFMSG command; however they always remain in the default class.

Message Filtering: Solicited Messages

Solicited messages are those generated as a result of VTAM commands. VTAM commands can be sourced from a system console or issued by OCS operators or NCL procedures. These messages always pass through message definition table (DEFMSG) processing. Only those messages indicating PPOPROC delivery in the DEFMSG table will be passed to PPOPROC for processing.

Note

Because they are normally displayed to the operator making a specific request, these messages are never delivered to other LOCAL receivers.

Message Filtering: Unsolicited Messages

Unsolicited messages are generated by VTAM to notify of an unexpected status change in the network—that is, a change which has not resulted from operator action.

Unsolicited messages are only delivered to the destinations specified for the VTAM message numbers in the DEFMSG table. If no delivery options have been set for an unsolicited message number, the delivery options for the unsolicited class entry are used, which, by default, are not set. To specify the delivery options, issue a DEFMSG UNSOL DELIVER=*delivery* command.

Designing a PPOPROC Procedure

A PPOPROC procedure should be written as a closed loop. When invoked, it processes until the first &PPOREAD statement is detected and then suspends processing until a message arrives.

As each message arrives, the procedure logic should branch to an appropriate processing point for that message. After processing the message or a group of associated messages, the procedure should then return to the initial &PPOREAD where processing is suspended once again.

PPOPROC should be designed to START worker processes to analyze any serious events reported and initiate recovery procedures so the main PPOPROC process remains free to continue monitoring the unsolicited VTAM message flow.

Warning

Be particularly careful when using verbs that are dependent on events outside PPOPROC control (such as &INTCMD followed by &INTREAD) as these can tie up PPOPROC processing and result in long PPO message queues.

Note

If PPOPROC does end, an NRD message is sent to all monitor status OCS operators, and normal PPO processing resumes.

Messages from PPOPROC

Messages from PPOPROC, including those issued by &WRITE statements or resulting from commands executed from within PPOPROC, (unless &INTCMD is used) are displayed, prefixed with a P, at all monitor status terminals.

PPOPROC Statements

Special processing verbs are provided for PPOPROC use:

- &PPOREAD retrieves the next PPO message for processing
- &PPOCONT returns a PPO message for normal delivery
- &PPODEL deletes a PPO message
- &PPOREPL replaces PPO message text
- &PPOALERT generates a message to send to PPOPROC

How to use these verbs is described in the *Network Control Language Reference*.

Multiple occurrences of any of these statements simplifies the processing of group messages, as the procedure can be structured for each type of message being processed.

In addition, PPOPROC can start worker processes to gather other relevant network information to act on PPO messages.

Testing PPOPROC

The SYSPARMS PPOPROC command is used to invoke PPOPROC, and can also be used to flush it. This allows a new copy of the PPOPROC procedure to be invoked from a subsequent SYSPARMS command.

The &PPOALERT statement lets you generate test PPO messages which can be used to check the logic of PPOPROC without having to wait for a real occurrence of the message.

Note

The base PPOPROC procedure never uses a preloaded copy of the procedure, so it is unnecessary to use the SYSPARMS UNLOAD command. Other procedures invoked from within PPOPROC will observe the normal preload conventions.

PPOPROC Prerequisites

- Before Management Services can receive messages from VTAM, a PPO START command must be successfully executed to authorize the system for the receipt of unsolicited VTAM messages. The PPO START opens the VTAM ACB specified on the SYSPARMS PPOACBNM= to gain access to VTAM messages. The STATUS command can be used to determine whether the system has successfully executed a PPO START command.
- To start your PPOPROC, specify SYSPARMS PPOPROC=*procname*.
- If you want to receive copies of VTAM commands in your PPOPROC, you must have SYSPARMS PPOSOCMD=PPOPROC specified and ensure PPOLOG=YES is specified in your VTAM startup.
- To receive specific messages, issue DEFMSG DELIVER=PPO commands either in your PPOPROC, or before starting it.
- PPO message delivery to PPOPROC will begin once the first &PPOREAD has been issued.

The SYSPARMS command is documented in the *Management Services Implementation and Administration Guide*. The PPO, STATUS, and DEFMSG commands are described in the *Management Services Command Reference*.

Intercepting OCS Messages (MSGPROC Procedures)

Management Services can intercept all messages being sent to an OCS screen or background environment.

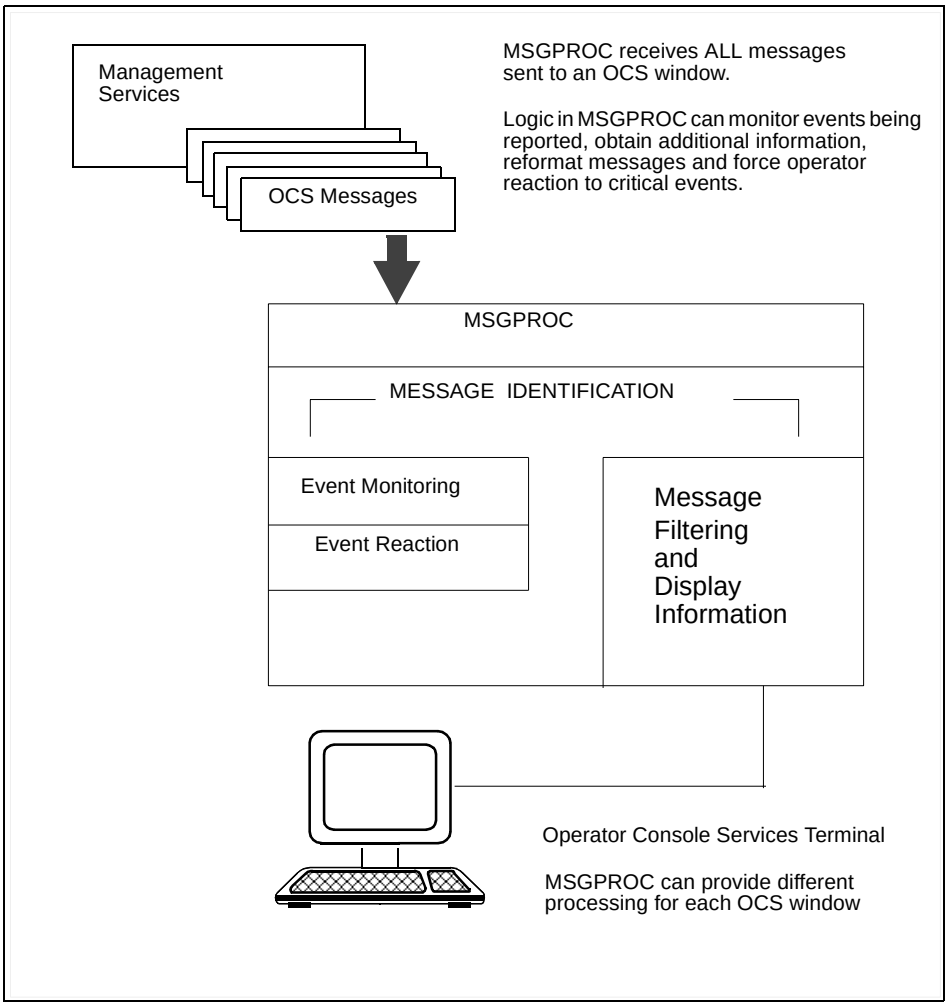
This facility is supported by a procedure known as MSGPROC. Before this facility can be utilized, a MSGPROC must be defined for each user ID requiring message interception.

To do this, update the user ID definitions (using UAMS, or an external security system) to define the name of the MSGPROC to be invoked.

The name provided in the MSGPROC field of the user ID definition is the name of a procedure in the procedure library. Different user IDs can have different MSGPROCs, while some user IDs might have no MSGPROC at all. You can change or flush the MSGPROC value with the PROFILE command.

Figure 10-4 shows a schematic of MSGPROC operation.

Figure 10-4. MSGPROC Monitors Each OCS Window Locally



Note

MSGPROC is supported for OCS mode and INMC users with ROF sessions, including background user IDs and console user IDs. However, you cannot use MSGPROC in dependent processing environments associated with &INTCMD use.

MSGPROC Statements

Special processing verbs are provided for MSGPROC use:

- &MSGREAD requests that the next message be made available
- &MSGCONT returns a message for normal delivery
- &MSGDEL deletes a message
- &MSGREPL replaces the text of a message

How to use these verbs is described in the *Network Control Language Reference* manual.

Multiple occurrences of any of these verbs simplifies the processing of group messages, as the procedure can be structured for the type of messages being processed.

In addition, MSGPROC can utilize &INTCMD processing facilities to perform other functions while processing a message.

Note

A MSGPROC procedure receives all messages that appear on your OCS window. It can therefore be used for a variety of purposes, such as acting on unsolicited messages received on MAI sessions, reformatting VTAM messages, or for recognising events and prompting an operator to take appropriate action.

Operators with suitable authority can change the name of their MSGPROC procedure or flush it, using the PROFILE command.

Designing MSGPROC Procedures

A MSGPROC procedure should be written as a closed loop. When invoked, it processes until the first &MSGREAD statement is detected again and then suspends processing until a message arrives for your terminal.

As each message arrives, the procedure logic should branch to an appropriate processing point for that message, or issue an &MSGCONT if the message is of no interest. After processing the message or a group of associated messages, the procedure should then return to the initial &MSGREAD where processing is suspended once again.

If the logic of the MSGPROC procedure allows it to terminate, an error message is issued and normal message processing resumes.

MSGPROC can also use the START command to create worker processes which process events or message groups asynchronously. This lets MSGPROC continue monitoring the flow of OCS window messages.

Messages from MSGPROC

If a user ID is defined with a MSGPROC, all messages destined for that user's OCS window are directed to the MSGPROC for processing. Messages that come from within the MSGPROC itself (such as those from &WRITE statements, or comments written to the user's terminal), are not passed to the MSGPROC for processing. This prevents recursive looping.

Testing MSGPROC

MSGPROC is invoked when you enter OCS mode. MSGPROC processes until the first &MSGREAD statement is encountered. The procedure remains active until terminated by an error (in which case normal message delivery resumes), or until you exit OCS mode. Exiting OCS mode flushes the MSGPROC procedure.

Subsequent re-entry to OCS mode will invoke the latest copy of the procedure, unless the procedure has been preloaded. If the procedure has been preloaded, you must first unload it using the SYSPARMS UNLOAD command and PRELOAD the replacement procedure.

MSGPROC Examples

The system is distributed with a MSGPROC example. Before writing your own MSGPROCs you should review $MSGPROC in the distribution library.

Define a temporary user ID where $MSGPROC is defined as the required MSGPROC, then log onto this user ID and enter OCS mode where the MSGPROC will be invoked.

$MSGPROC intercepts VTAM display commands and provides a single line summary. It extracts information from various lines generated by the VTAM command, analyzes them within the procedure, and generates a response to the operator. The example might need tailoring for your VTAM level.

User ID Considerations for System Level Procedures

All NCL processes execute on behalf of an authorized Management Services user ID. Private users who log on to Management Services and execute NCL processes, execute their NCL processes within their own NCL processing regions.

System level procedures also require an NCL processing region for execution. The principal procedures, LOGPROC and PPOPROC execute under special internal user ID environments. LOGPROC executes under a special user ID, and PPOPROC executes under a special PPO interface user ID.

The internal user IDs providing the execution environments for these system level processes are regarded as standard users by Management Services. They are logged on in the standard way and can have UAMS (or security exit) user ID definitions. If you want to profile these internal user IDs in special ways, you can define them in the same way as real users.

If remote operator functions are needed by processes such as LOGPROC, the user ID for the log user (the background logger) must be defined for any remote SOLVE systems that will be sent commands, otherwise the ROF connection is refused.

11

Using Advanced Program-to-Program Communication (APPC)

This chapter describes the programming facilities provided by Management Services' Advanced Program to Program Communication (APPC).

This chapter discusses the following topics:

- Management Services APPC
- APPC Conversations
- Conversation Allocation
- Attaching a Procedure
- Send Operations
- Receive Operations
- Error Processing
- Conversation Deallocation
- &APPC Return Code Information
- Application Design

Management Services APPC

SNA APPC is a high-level application programming interface that allows program-to-program communications and the development of distributed applications between network nodes that support Logical Unit type 6.2 (LU6.2). LU type 6.2 communication facilities form the basis for SAA's Common Programming Interface for Communications (CPI-C).

In the Management Services implementation of APPC, the high-level application programming interface is provided by the NCL &APPC verb which provides access to a full set of LU6.2, or APPC, programming capabilities.

These facilities let an NCL procedure communicate with another APPC application in a structured manner as defined by the LU6.2 protocol. The partner application can exist within the same, or a remote SOLVE system, or any other application system that supports LU6.2 protocols.

In addition, Management Services APPC supports a number of extensions that assist the development of client/server applications within NCL or spanning other APPC platforms.

Before writing NCL procedures that use these APPC facilities, please familiarize yourself with the fundamental APPC concepts and implementation procedures described in the chapter titled *Using APPC to Communicate With Other Systems* in your *Management Services Implementation and Administration Guide*. In addition, see the *SNA Transaction Programmer's Reference Manual for LU Type 6.2*, which is the authoritative source for detailing the LU6.2 verb set.

APPC Conversations

NCL procedures (and transaction programs in general) communicate by establishing a communication path called a conversation. Conversations use LU6.2 sessions to exchange data and protocols between the communicating procedures. All conversations consist of three main phases:

- Conversation initiation (allocation and attach processing)
- Data exchange (sending and receiving data operations)
- Conversation termination (deallocation processing)

Once a conversation is allocated to a session, a send-receive relationship is established between the participating programs. Initially, the procedure that requested the conversation is allowed to issue send data verbs while the other procedure issues receive data verbs. This send-receive relationship can change many times during the life of the conversation.

To terminate the conversation the procedures request deallocation processing, by issuing the appropriate deallocate verb. The following sections give a detailed description of the above conversation phases and the associated verbs.

Note

The Management Services implementation of APPC allows NCL procedures within the same SOLVE system to communicate, without the need to use SNA sessions. This is a highly effective means of data transfer between NCL processes.

The LU6.2 Verb Set

The LU6.2 verb set defines a structured means of communication between two programs. A strict protocol exists at many layers in the LU6.2 implementation, including the definition of the application verb set and conversation states. Most APPC requests are handled by the NCL &APPC verb, while conversation state and other status indicators are accessible to NCL through a range of system variables.

The &APPC Verb

All actions on a conversation are supported through the &APPC verb. The specific request is identified by the keyword immediately following the &APPC verb. In general, the keyword identifying each request corresponds closely to the LU6.2 architected verb syntax such that the specific LU6.2 architected verb is apparent from the NCL syntax. The relationship between the NCL options and the LU6.2 architected verb set is described in the reference section for the particular &APPC request. The set of &APPC requests is listed below:

- &APPC ALLOCATE_DELAYED
- &APPC ALLOCATE_IMMEDIATE
- &APPC ALLOCATE_NOTIFY
- &APPC ALLOCATE_SESSION
- &APPC CONFIRM
- &APPC CONFIRMED
- &APPC DEALLOCATE
- &APPC FLUSH
- &APPC PREPARE_TO_RECEIVE
- &APPC RECEIVE_AND_WAIT
- &APPC RECEIVE_IMMEDIATE
- &APPC RECEIVE_NOTIFY
- &APPC REQUEST_TO_SEND
- &APPC SEND_DATA

- &APPC SEND_ERROR
- &APPC TEST

Note

Management Services does *not* support LU6.2 sync-point processing, and hence the verbs and states that implement this are unsupported.

Conversation States

Conversations are managed by transition through a number of states. The state of a conversation is reflected in the &ZAPPCSTA system variable, and can be one of the following:

- RESET
- SEND
- RECEIVE
- DEFER_RECEIVE
- DEFER_DEALLOCATE
- CONFIRM
- CONFIRM_SEND
- CONFIRM_DEALLOCATE
- DEALLOCATE

Conversation Processing

Only a small number of verbs are acceptable to NCL from any given conversation state, otherwise a state error ensues. Following the successful completion of each verb, either the state remains unchanged, or a single state transition is possible. This means that the programmer can always determine the next course of action for the conversation. If verb completion is not successful then further analysis of the reason, by examination of conversation status fields, might be required to determine the most appropriate course of action to follow.

In the simplest analysis, the APPC protocol is a flip-flop communication channel where, in normal operation, the procedure currently sending controls the data flow. Hence, the *first speaker* procedure continues sending data until it decides to stop. It might decide to stop only when all data has been sent. For example, after sending a request for a database query, in order to reverse the direction of data flow and receive the response. Or it might pause during transmission to request confirmation (actual receipt by the partner procedure) of that portion of the data sent so far.

Only when the current speaker changes direction can the partner procedure send data the other way. However, if some condition arises that is generally unrecoverable, an error indication can be sent to the current sender to cause transmission to cease, and allow the current receiver to enter send state.

Within Management Services all information being sent is buffered until enough data is accrued to warrant transmission, or the need for some application response dictates that an actual network transmission take place. This provides for efficient use of network resources, but means that in order to synchronize local and remote processing, the application must manage the protocol appropriately. This is achieved through use of the confirmation protocols discussed later in this chapter.

Return Codes and System Variables

Following each verb the &RETCODE and &ZFDBK system variables are set to indicate the success or otherwise of the &APPC request. They are documented later in this chapter.

In addition, for each conversation, a set of system variables is maintained that provides information concerning the conversation status. These are listed in Table 11-1.

Table 11-1. APPC Conversation Status System Variables

&ZAPPCELM	message from an error log GDS variable
&ZAPPCELP	product set information from an error log GDS
&ZAPPCID	conversation identifier
&ZAPPCIDA	the conversation identifier for the transaction that started the NCL process
&ZAPPCLNK	local link name
&ZAPPCMOD	session mode name
&ZAPPCQLN	network qualified local luname
&ZAPPCQRN	network qualified remote luname
&ZAPPCRM	current receive map name
&ZAPPCRPI	received protocol indicators
&ZAPPCSM	current send map name
&ZAPPCSTA	conversation state
&ZAPPCSYN	conversation sync_level
&ZAPPCTYP	conversation type
&ZAPPCWR	what_received indicator
&ZAPPCWRI	what_received short indicator
&ZAPPCRTS	request_to_send indicator
&ZAPPCTRN	transaction identifier

Conversation Allocation

An executing procedure establishes a communication path, termed a conversation, by the process known as allocation. An allocation request is deemed to take place from reset state, and can be issued as one of the following verb options:

- &APPC ALLOCATE_SESSION (or just &APPC ALLOCATE)
- &APPC ALLOCATE_DELAYED
- &APPC ALLOCATE_IMMEDIATE
- &APPC ALLOCATE_NOTIFY

The Transaction Identifier

One of the parameters supplied on the allocate verb is the TRANSID (transaction identifier) which directs Management Services to the APPC Transaction Control Table, or TCT. An appropriate TCT entry for the transaction must be located or the conversation fails. The TCT entry is used to verify the request before further information is extracted to complete the request if necessary.

Destination Selection

The TCT entry can contain default destination information, by way of a link name or network LU name, to be used for the request. However the LINK or LUNAME parameters on the allocate verb can be used to override any TCT destination information (and will be required if the TCT has no default destination information).

Once a destination has been isolated resources are allocated to set up the communication path, and eventually a procedure or program is attached in the target system to service the request.

If the target system is the local SOLVE system an NCL procedure is started as an *attached* procedure. Such procedures are generally written to perform a specific function, or possibly a range of functions, on behalf of the invoking procedure.

If the target system is outside of the local SOLVE system, an SNA LU6.2 session connecting the remote destination must be located. This session is then allocated to the conversation for the conversation duration. The fact that a session is interposed between the communicating procedures is completely transparent to the allocating procedure.

Allocation and Sessions

To complete an allocation to a remote destination a session is required. If no session to that destination currently exists, then for all requests except &APPC ALLOCATE_IMMEDIATE, an attempt will be made to establish an APPC link to the remote LU. Other than the fact that it is an automatic link activation request, the result is no different from an operator starting a link by command.

For the &APPC ALLOCATE_SESSION and ALLOCATE_NOTIFY requests, the verb will not complete until a session can be assigned for the conversation. This can involve waiting for new sessions to be established, or waiting for active conversations to end and free up a session.

For the &APPC ALLOCATE_DELAYED request, a delayed session assignment is allowed. The verb completes as though a session can be assigned and processing continues. However, if transmission is actually required at some stage, the procedure will be suspended until session assignment can be performed.

The &APPC ALLOCATE_IMMEDIATE request only completes successfully if a session can be guaranteed immediately for transmission.

Setting Program Initialization Parameters

User data can be passed as parameters on the allocate verb. These can be in the form of one or more NCL tokens. Each NCL token passed as a parameter on the allocation will appear as a separate parameter in the remote end. These parameters are available to the attached procedure as *program initialization parameters*, described in the next section.

Allocation Completion

After a successful allocation, the procedure is placed in send state and the &ZAPPCID system variable is set to provide the identifier of the conversation created. More than one conversation can be allocated and operated concurrently by a single NCL procedure.

Note

The allocation can complete without any transmission taking place, and hence the procedure enters send state before the target procedure is invoked to service the request.

Attaching a Procedure

At some time following allocation, data is actually transmitted to the target system in the form of an attach request.

When an attach request is received by Management Services, the TCT is examined to locate the procedure that will service the request. The procedure is said to be *attached* and a copy of the procedure is invoked as a new NCL process.

Client/Server Terminology

Client/server processing lends itself well to the use of APPC as a protocol. For this reason, the term *client* is often used to mean the allocating procedure, and *server* to mean the attached procedure. Many of the &APPC verb extensions described in Chapter 12, *SOLVE APPC Extensions*, are designed to support the client/server model.

Execution Environment

If the conversation is executed as an unsecured transaction, the procedure will execute in the background server (BSVR) environment. Otherwise an APPC user region is located (or created and signed on if none exists), for the user ID carried in the attach request, and the procedure started within that user region. When no further NCL activity exists in an APPC user region it is signed off and deleted.

Accessing Program Initialization Parameters

Once attached, the NCL procedure begins execution as any other NCL procedure, except that it already has an active conversation.

If program initialization parameters were passed on the allocation request, they are accessible as the standard NCL parameters &1 &2 &3through to &*n* in the usual manner. Each token represents a single passed parameter and hence the data available is limited to the current maximum token size.

Attach Processing

The conversation that attached the procedure is identified by the &ZAPPCIDA system variable and is also the current conversation identified by the &ZAPPCID system variable. The procedure is in receive state on this conversation. An attached procedure has access to all the usual NCL facilities and can allocate further conversations if desired.

Send Operations

Data can be sent on a conversation only from send state. Send state is entered in one of the following ways:

- Automatically following a successful allocation
- At any time when the remote conversation partner that is currently sending decides to stop sending and enter receive state, thereby placing the local end in send state.
- After issuing a SEND_ERROR which forces send state locally

Sending Data

The only verb used to send data is:

```
&APPC SEND_DATA
```

Data can be in the form of one or more NCL tokens, or it can be a Mapped Data Object, or MDO. Even where multiple tokens are identified on the send operation, they are packaged as a single unit of transmission, called a GDS variable, to the remote end where they will eventually become available from a single receive request.

There is no limit to the number of variables or MDOs, or total size of the GDS variable used for transmission.

Following a successful send data operation the conversation remains in send state. Otherwise the return code information should be examined for details on the send failure.

Data Mapping Support

APPC can send a map name of up to 64 characters as control information when data is sent. This map name is used by the remote system to interpret the contents of the data transmitted and indicates how it should build any local data structures.

Data Mapping for NCL Tokens

When sending NCL tokens, a map name of $NCL is sent by default with the data to indicate that the data is comprised of one or more NCL tokens. If the receiving system understands the $NCL structure, it can decompose the datastream into the individual tokens sent. In this way NCL allows a single send to specify multiple tokens as a range or list, and in the target system a single receive can re-create such a range or list. For example, the request:

```
&APPC SEND_DATA VARS=A* RANGE=(1,10)
```

can be sent, and satisfies a remote request:

```
&APPC RECEIVE_AND_WAIT VARS=B*
```

such that the variables B1,B2,...,B10 in the remote system are created with the same values as A1,A2,...,A10 in the sending system.

You can override the map name sent. However, if a map name other than $NCL is specified the tokens are not structured as above for output, but are simply concatenated together to form the single GDS variable which is the unit of transmission. When communicating with a non-NCL system, this may let you construct the appropriate pieces of the data in separate tokens before sending them as a single datastream. The interpretation of the block of data received in the remote system is then implied by the map name sent.

Data Mapping and Mapping Services

When sending data from an MDO, the MDO name specified determines the length of the data unit to be sent. The map name sent by default is the fully qualified map name hierarchy as defined to Mapping Services.

For example, if a CNM alert record was in an MDO named NEWS, mapped by the map named CNM, then the second Product Set Identifier sub-vector within that record might be referenced from NCL as:

```
NEWS.NMVT.ALERT.PSID{2}
```

If the Product Identifier sub-vector within it was the subject of a send operation it can be sent by:

```
&APPC SEND_DATA MDO=NEWS.NMVT.ALERT.PSID{2}.PRODID
```

However, the fully qualified map name hierarchy for the object is:

```
CNM.NMVT.ALERT.PSID.PRODID
```

as defined to Mapping Services, and this will be the map name sent. If the target system understands this mapping hierarchy it will reconstruct the MDO and map it accordingly. For example a receive request of:

```
&APPC RECEIVE_AND_WAIT MDO=NEWSREC
```

would result in the object being:

```
NEWSREC.NMVT.ALERT.PSID.PRODID
```

Note

All name instances are now implied to be 1, as only the second PSID instance was selected for the send. No data existing in any of the elements of the intervening name segments is sent, however the qualified names are sent to provide the context of the actual data transmitted.
If the entire MDO had been sent, for example:

&APPC SEND_DATA MDO=NEWS

then both PSIDs would be received and the entire NEWSREC structure received would be identical to the NEWS structure sent.
Again, you can choose to override the map name sent, allowing the target system to implement data interpretation independently.

Sending Data When Data Mapping Is Not Supported

Not all LU6.2 systems support data mapping. If data mapping is not supported (as defined in the Option Set Control Table selected for the destination system) then no map names are sent and all data sent is always transmitted *as is*.

If a number of tokens are specified on the SEND_DATA request then they are simply concatenated together to form the single unit of data transmission. If an MDO is sent then the data from the MDO comprises the entire data unit. The structure of this data unit, and its interpretation in the remote system are the responsibility of the application developer.

Requesting Confirmation of Data Sent

While in send state a procedure can request confirmation of any data sent. This is achieved by the verb:

```
&APPC CONFIRM
```

Processing is suspended until the remote procedure has received all data sent up until that point and receives the confirmation request. The normal response is:

```
&APPC CONFIRMED
```

This response indicates that all data was received and processed normally. The receipt of the confirmed response satisfies the confirm request and allows the sending procedure to continue processing in send state.

Forcing Data Transmission

If deemed necessary by the procedure it can, while in send state, force the transmission of any data buffered by APPC with the request:

```
&APPC FLUSH
```

All queued data is scheduled for transmission by this request, however it is usually unnecessary as data will be transmitted as it accumulates during normal send operations. When a reasonable amount of data is buffered, or if requests which require some response (such as an &APPC CONFIRM) are issued, then a transmission is scheduled automatically by the system. A flush operation does not alter the state.

Switching State from Send to Receive

When a procedure has completed all send operations and expects some response data from the conversation partner it can switch from send to receive state by issuing:

```
&APPC PREPARE_TO_RECEIVE
&APPC PREPARE_TO_RECEIVE TYPE=FLUSH
&APPC PREPARE_TO_RECEIVE TYPE=CONFIRM
```

If no type is specified then it will default to FLUSH if the conversation has a sync_level of NONE, or CONFIRM if the conversation has a sync_level of CONFIRM (as set by the allocation request and reflected in the &ZAPPCSYN system variable).

The FLUSH option simply ensures all data is sent to the other end with a request to change direction. For the CONFIRM option the remote procedure must issue an &APPC CONFIRMED request before the verb completes and receive state is entered.

If data is expected in response then a confirmation can usually be avoided by simply issuing, from send state, either request:

```
&APPC RECEIVE_AND_WAIT
&APPC RECEIVE_NOTIFY
```

These requests will perform an implied &APPC PREPARE_TO_ RECEIVE TYPE=FLUSH before placing the procedure in receive state.

Receive Operations

Data can be received on a conversation only while in receive state. Receive state is entered in one of the following ways:

- Automatically when a procedure is attached by an allocation request
- At any time when the procedure decides to stop sending and enter receive state voluntarily
- When an error is received from the remote system, forcing the local procedure into receive state

Receiving Data

The verb options available for requesting received data are:

```
&APPC RECEIVE_AND_WAIT
&APPC RECEIVE_IMMEDIATE
```

while an asynchronous notification that data has been received (without actually receiving it into target variables) can be requested by:

```
&APPC RECEIVE_NOTIFY
```

Following notification, the RECEIVE_IMMEDIATE request can be used to retrieve the data. When data is received, it can be placed in one or more tokens, or into an MDO.

Following a receive operation the &ZAPPCWR and &ZAPPCWRI indicators are set explaining the nature of the data received. It is possible to get a successful receive (&RETCODE 0) when no data is placed into the target variables. However the what-received indicator could show that some request, such as a confirm, has been received and hence the procedure should issue the &APPC CONFIRMED response.

Receiving Data into NCL Tokens

Either a list or range of NCL tokens can be specified as the target variables for a receive operation. For example:

```
&APPC RECEIVE_AND_WAIT VARS=$* RANGE=(1,20)
&APPC RECEIVE_AND_WAIT VARS=(A,B,C,D)
```

The contents of each variable will be set depending upon the data mapping for the conversation. If no mapping is supported, or if the map name is other than $NCL, then the received GDS variable (which is the unit transmitted) is segmented according to the maximum token size by default. You can specify individual segment sizes when using the list form by specifying a parenthesized integer after each variable name, for example:

```
&APPC RECEIVE_AND_WAIT VARS=(A(10),B(8),C(4),D(32))
```

If mapping is supported and the map name is $NCL, then the received GDS variable, which is the unit transmitted, is assumed to be correctly formatted by the sending NCL system. Its structure maintains the contents of the individual tokens that were specified on the send operation, and the target variables are reconstructed identically.

In any case, no data transformation takes place. If the transmitted data contains undisplayable characters, subsequent processing might require its conversion to hexadecimal characters (through the &HEXEXP built-in function). This is your responsibility.

If insufficient variables are supplied to receive all the data transmitted in the data unit, the residual data is lost. Note that the map name received is available in the &ZAPPCRM system variable.

Receiving Data into an MDO

As an alternative to the use of NCL tokens an MDO can be specified as the target for a receive operation, for example:

```
&APPC RECEIVE_AND_WAIT MDO=APPCREC
```

The MDO is structured depending upon the data mapping for the conversation. If no mapping is supported, or if the map name is unknown to Mapping Services, then the data from the received GDS variable which is the unit transmitted forms the entire contents of the MDO. The MDO is unmapped, but the MDO name refers to the entire data transmitted.

If mapping is supported and the map name is known, then the data from the received GDS variable which is the unit transmitted, is assigned into the MDO, specified according to the map name received. If the map name consisted of more than one name segment, then the first name segment is assumed to be the actual map name, and the remaining segments qualify the data in the usual Mapping Services manner.

For example, if the map name received was:

```
CORPORATE.PAYROLL.EMPLOYEE
```

and the receive specified was:

```
&APPC RECEIVE_AND_WAIT MDO=USER
```

then the MDO structure named USER.PAYROLL.EMPLOYEE and mapped by the Mapping Services map name CORPORATE, is assigned the received data. However if the map name CORPORATE was unknown to Mapping Services then the MDO structure named USER contains all the data and it is unmapped.

Note again that you can determine the map name received through the &ZAPPCRM system variable and process the MDO contents accordingly. For example, if a transmitted map name is unknown to Mapping Services, or is an invalid Mapping Services map name, the data can be placed into an unmapped MDO. The map name is then examined and the MDO assigned to a new structure with Mapping Services mapping through the &ASSIGN verb.

Responding to a Confirmation Request

Following a receive operation, the &ZAPPCWR might indicate that a confirmed response is required if the &ZAPPCWR value is:

```
CONFIRM
CONFIRM_SEND
CONFIRM_DEALLOCATE
```

in which case the procedure can respond:

```
&APPC CONFIRMED
```

after which receive state, send state, or deallocate state is entered respectively.

Receiving a Send Indication

Following a receive operation, the &ZAPPCWR can indicate that the local procedure has entered send state if the &ZAPPCWR value is SEND. This indicates that the remote end has completed its send operations and has entered receive state.

Receiving a Deallocation Indication

Following a receive operation, the &ZAPPCWR can indicate that the conversation has been terminated unconditionally by the remote procedure. In this case &RETCODE 4 is returned, and the &ZAPPCWR value is DEALLOCATE, the only allowable action is to issue:

```
&APPC DEALLOCATE TYPE=LOCAL
```

after which all conversation information is removed from the system.

Error Processing

Either conversation partner can send an error indication at any time by issuing &APPC SEND_ERROR.

When issued from send state, the remote end (in receive state) will get a return code indicating program_error_no_truncation, and &RETCODE is 8. This indicates that the error sent did not cause any loss of data. No state changes occur.

When issued from receive state, the remote end (in send or receive state) will get a return code indicating program_error_purging, and &RETCODE is 8. This indicates that the receiver is purging all subsequent data sent and the sender has an obligation to enter receive state (if it has not already done so).

The effect of a SEND_ERROR is to halt communication and place the error sender in send state. However it should be used with discretion as it is most disruptive of normal data flows. Usually a reason for the error will follow as a normal data send operation.

Some errors can be detected by the APPC Services layers, and will appear as svc_error_purging and so on. For example, if a procedure which is one end of a conversation terminates abnormally, a deallocate abend condition is raised and an error notification is sent to the other partner. A message accompanies such a condition and will appear in the log of the remote system. It can be accessed through the &ZAPPCELM system variable before the &APPC DEALLOCATE TYPE=LOCAL statement is issued.

Conversation Deallocation

Regardless of which end started a conversation, any procedure can terminate it from send state by issuing one of the deallocate requests:

```
&APPC DEALLOCATE
&APPC DEALLOCATE TYPE=FLUSH
&APPC DEALLOCATE TYPE=CONFIRM
&APPC DEALLOCATE TYPE=ABEND
```

If no type is specified, then it will default to FLUSH if the conversation has a sync_level of NONE, or CONFIRM if the conversation has a sync_level of CONFIRM (as set by the allocation request and reflected in the &ZAPPCSYN system variable).

A TYPE=CONFIRM deallocation is conditional upon the remote procedure issuing an CONFIRMED response. The only other valid response is a SEND_ERROR, which means the conversation remains active and the error sender is placed in send state. If &RETCODE of 0 is returned the deallocation was successful.

A TYPE=FLUSH or TYPE=ABEND is unconditional as long as the request is accepted. A &RETCODE of 0 indicates that the conversation is terminated.

Sample Conversations

A set of simple APPC conversations illustrating the use of the &APPC verb is provided in the distributed sample library. See Appendix E, *Sample APPC Conversations*, in this manual, for an explanation of how to set up and run the sample conversations.

&APPC Return Code Information

All &APPC verb options complete by setting a number of NCL variables with return code information.

General completion information is contained within the &RETCODE system variable, and is qualified by the &ZFDBK system variable. &RETCODE values of 0 and 4 occur in normal operation, while &RETCODE values of 8 or higher indicate an error condition. If an error is detected, the &SYSMSG user variable is set providing a explanation of the error condition.

A number of APPC system variables are available providing information about the current conversation being operated. For example, following a receive operation the &ZAPPCWR and &ZAPPCWRI (what-received indicators) provide information about what satisfied the receive.

See the &APPC verb description in the *Network Control Language Reference* for details of &RETCODE and &ZFDBK system variable settings.

Application Design

An important aspect of APPC programming is the consideration of application design. The protocol and verb set provided by APPC does not dictate a specific mode of use and can be flexibly adapted to a number of situations. In this sense, the application's use of the APPC verb set should be considered as part of the overall application design, and this further implies that the application spans the conversation.

Thus what is ostensibly a single application can be split into two (or more) procedures which communicate using the NCL &APPC verb. This form of application lends itself well to client-server type roles. See the section titled *APPC Client/Server Processing* in Chapter 12, *SOLVE APPC Extensions.* Commonly, the client procedure will act as the man-machine interface, providing the presentation aspects of the application.

The server procedure can deal with the data organizational aspects of the application, fetching and returning to the presentation procedure one or more records depending upon the request.

There are some important advantages in this approach:

- The code for handling the functional or presentation aspects is physically separate from the code for dealing with data organization aspects. This simplifies the processing required in each part, and assists future maintenance. For example, the server procedure could be completely replaced at any stage to handle a new database or file system.
- As the code is separate, the procedures can be executed in different systems without any change to the application. For example, the client end can always reside in a work station while the server end is in a host system. However, they can also execute within the same system. An additional advantage is that should a server procedure need to be moved to another system, for example if a database has been moved, then no code changes are necessary. Simple changes to the APPC Control Tables can be used to redirect transactions to the new system.
- Procedures are executed on demand. Rather than initializing a process to wait for work, the required procedure can be attached when a demand for its services is received.

12

SOLVE APPC Extensions

Management Services offers a number of extensions to assist in establishing communications using client/server processing concepts. These extensions assist in application development between SOLVE platforms, but might not be supported by other products which implement APPC.

This chapter discusses the following topics:

- SOLVE/APPC Extended Verb Set
- SOLVE/APPC Transactions
- APPC Client/Server Processing
- Server Processes
- Client/Server Connection Mode
- Transferring a Conversation

SOLVE/APPC Extended Verb Set

The following verbs are available in SOLVE/APPC:

- &APPC ATTACH_DELAYED
- &APPC ATTACH_IMMEDIATE
- &APPC ATTACH_NOTIFY
- &APPC ATTACH_SESSION
- &APPC CONNECT_DELAYED
- &APPC CONNECT_IMMEDIATE
- &APPC CONNECT_NOTIFY
- &APPC CONNECT_SESSION
- &APPC DEREGISTER
- &APPC REGISTER
- &APPC RPC
- &APPC SEND_AND_CONFIRM
- &APPC SEND_AND_DEALLOCATE
- &APPC SEND_AND_FLUSH
- &APPC SEND_AND_PREPARE_TO_RECEIVE
- &APPC SET_SERVER_MODE
- &APPC START
- &APPC TRANSFER_ACCEPT
- &APPC TRANSFER_CONNECT
- &APPC TRANSFER_REJECT
- &APPC TRANSFER_REQUEST

SOLVE/APPC Transactions

In addition to user-defined transactions SOLVE/APPC supports a number of system transactions that assist in NCL communication, especially in the area of client/server processing. These transactions are as follows:

START	This is used to initiate a new NCL process in either the local or a remote system
RPC	This is used to call a procedure in either the local or a remote system in the manner of a remote procedure call
ATTACH	Establishes a standard APPC conversation by attaching an NCL procedure by name, rather than indirectly through a transaction identifier
CONNECT	Establishes a conversation connection to an existing NCL process in either the local or a remote system.

These system transactions are automatically defined in the Transaction Control Table during system initialization, but can be modified by the installation if necessary using the REPTRANS, DELTRANS and DEFTRANS commands.

The START Transaction—Remote Process Start

SOLVE/APPC incorporates the system START transaction that allows any NCL procedure to be started as a new process through the &APPC START request. For example:

```
&APPC START PROC=proc DOMAIN=test ...
```

The system TCT entry for the START transaction is used to complete the conversation setup options, including the transaction security requirements.

Any procedure that is designed to be started via the usual START command can be started in this manner using SOLVE/APPC. The started procedure need not be aware that it was started using SOLVE/APPC, has no access to the APPC conversation that established the new process, and has no requirement to use any SOLVE/APPC facilities.

The new process can be created in any of the following ways:

- As a dependent process (of the requesting process) in the local SOLVE/APPC system
- As an independent process within the same region as the requesting process
- As an independent process within any APPC region (where authorized) in the local SOLVE/APPC system
- As an independent process within any APPC region (where authorized) in any connected SOLVE/APPC system

The initiating process can request an immediate indication of the success or failure of the new process creation. It can pass the usual procedure initiation parameters to the target procedure. In addition, any subset of NCL variables, or MDO data, can be copied from the environment of the initiating procedure to that of the new process. Once the request completes all links between the initiating process and the new process are lost, and the new process operates completely independently of the initiating process.

Note

When a process is started as a dependent process, the usual NCL relationships hold true.

The Remote Procedure Call (RPC) Transaction

SOLVE/APPC incorporates the system RPC transaction that allows any NCL procedure to be executed from the context of the calling procedure through the &APPC RPC request. For example:

```
&APPC RPC PROC=proc LINK=link23 PARMS=(A,B,C)
```

The system TCT entry for the RPC transaction is used to complete the conversation setup options, including the transaction security requirements.

Many procedures that are designed to be called from another procedure via the usual EXEC command can operate successfully in this manner using SOLVE/APPC. However, the only context that can be transferred between the calling procedure and the remote procedure, is in the form of NCL variables and MDO data. All other resources in the calling environment (such as files, internal read queues, and so on) remain the strict domain of the calling procedure. The called procedure need not be aware that it was started using SOLVE/APPC, has no access to the APPC conversation that established the call path, and has no requirement to use any SOLVE/APPC facilities.

The target procedure is started as an independent process in the same or any connected SOLVE/APPC system, with the context passed. The calling procedure is suspended until the remote procedure completes its execution and terminates. However, the called procedure need not be aware that it was invoked via a remote procedure call.

When the remote procedure terminates, control is returned to the calling procedure with an indication of the success or failure of the request. Shared NCL variable information can be passed back on return. If the called procedure terminates abnormally, or APPC communication is lost to a remote system, the appropriate error information is returned.

The new process can be created in any of the environments allowed for a remote process start.

The ATTACH Transaction—Allocate a Procedure

SOLVE/APPC incorporates the system ATTACH transaction that allows any NCL procedure to be attached and establishes the usual APPC conversation connection. For example:

```
&APPC ATTACH PROC=proc DOMAIN=test
```

This form of allocation proceeds as a special internal transaction. It sets up the conversation with the nominated procedure without the requirement to set up a Transaction Control Table entry for the procedure. The system TCT entry for the ATTACH transaction is used to complete the conversation setup options, including the transaction security requirements.

Note

This form of allocation can be useful between SOLVE systems where the procedure name is obtained indirectly, but should not be thought of as a general replacement for the use of the Transaction Control Table.

Once started, the conversation is available in the attached procedure, exactly as it would had a standard allocation request been issued. The conversation is operated by both partners in the usual manner.

The CONNECT Transaction—Connect to an Active Process

SOLVE/APPC incorporates the system CONNECT transaction that requests a connection between the a client procedure and any other active NCL process to act as a server. It establishes the usual APPC conversation connection. For example:

```
&APPC CONNECT NCLID=nclid DOMAIN=test
```

This form of allocation proceeds as a special internal transaction, and sets up the conversation between the client and the nominated server process if permitted by the server. The system TCT entry for the CONNECT transaction is used to complete allocation options.

Note

The process must be active or the conversation will terminate with an allocation failure.

Once connected, the conversation is available in the target procedure exactly as it would had a standard allocation request been issued. The conversation is operated by both partners in the usual manner.

APPC Client/Server Processing

APPC is a general application protocol for communication between any two programs in an SNA environment. It can be used to deliver data in substantially one direction—such as a file transfer. It can be used for a complicated dialog—such as a terminal to application datastream. However, it can be easily adapted to implement communication along the client/server processing model.

The usual model of client/server processing is that there are potentially many clients, and usually a smaller number of servers, in a distributed processing environment. The clients request information and they are served that information by a server. A simple question, followed by one or more related responses, is an example of this approach, however the actual dialog could be more complicated.

Ideally the client does not nominate the server, but merely requests a particular type of service, and depending upon configuration options, the transaction will be directed to an appropriate server for processing.

Server Processes

A server is a single NCL process that can accept connections from one or more clients, either serially or concurrently, at the server's discretion. Registration of the server is successful if the server name is unique within some scope, as determined by the server name registration request. Registration of the server name can be tied to process creation such that if the registration is unsuccessful the process creation fails. Alternatively, an executing process can attempt to register itself as a server at any time.

A transaction that starts an NCL procedure can now indicate, through the Transaction Control Table, that, once attached, the target NCL procedure is to be registered as a server process. A server process can also be started by the usual START command before any communication is necessary.

In general, any NCL process, regardless of whether or not it has a registered server process name, can in fact behave as a server process. That is, any active NCL process can accept client connections. Server name registration provides a mechanism for preventing duplication of server processes, but, more importantly, assists in targeting the correct server by supplying a meaningful name. While a server is active, new transactions can target the process and request connection. Such transactions can, at the server's discretion, be queued to the server through the SOLVE/APPC transfer mechanism, or automatically connected to the server for immediate operation.

Any process that has a registered server name can be targeted, by defining transactions in the TCT that allocate that particular server name. If the server is not active, the first transaction selecting a TCT entry for that server name will start the server process. Subsequent transactions locate the active server process and queue a connection request, as described below. Any user defined transaction can target a server in this manner. In addition, the ATTACH and CONNECT transactions allow a server to be targeted by the requestor.

Client/Server Connection Mode

During server process creation and initialization, all client connection requests that target the process remain pending until the server declares its operational mode for client connection. It can choose to automatically accept new connections, request notification before accepting connections, or reject connections.

Connections only apply to new conversations targeting an active process such as:

- Any standard transaction where the TCT entry indicated a server name as target, and the server is active
- Any ATTACH transaction that targets a server name, and the server is active
- Any CONNECT transaction that always targets an active NCL process

However an explicit conversation transfer request, resulting from an &APPC TRANSFER operation, is not considered a connection request and its mode of operation is unchanged.

The connection mode chosen by the server depends upon whether it intends to serialize connection processing, or operate multiple conversations concurrently. It can also depend on whether the server makes any use of the PIP data sometimes present with new conversations.

Automatic Connection Mode

The server declares it will operate in automatic connection mode by issuing the following statement after process initialization:

```
&APPC SET_SERVER_MODE CONNECT=ACCEPT
```

or

```
&APPC REGISTER SERVER=server CONNECT=ACCEPT
```

Following this statement any pending connection requests, or any subsequent connection requests, are automatically connected to the server and available to operate immediately in receive state. When using this mode no access to the PIP data, if present, for a new conversation is available. Hence this mode of operation is unsuitable for servers that service transactions making use of PIP data.

Since all new conversations connect in receive state, after an automatic connection they would satisfy a receive request that specifies any active clients. For example:

```
&APPC RECEIVE_AND_WAIT ID=CLIENTS VARS=A*
```

or

```
&APPC RECEIVE_NOTIFY ID=CLIENTS
```

Any conversation that connected to the server, that is any client conversation, can satisfy such a receive. However, conversations started by the server, even if they are in receive state, do not satisfy these requests. In either case a new connection, or an existing conversation where more data had just arrived, could satisfy the receive request.

Conversations are serviced in the order that data arrives. The actual conversation satisfying the receive is identified by the usual &ZAPPC*xxx* system variables. The server, having received some data, can choose to operate that conversation specifically to satisfy the client, before issuing the generic receive against all clients to process the next item of work, thus serializing server activity.

Notification Mode

As an alternative, the server can declare it will operate in notification mode by issuing:

```
&APPC SET_SERVER_MODE ...  CONNECT=NOTIFY
```

or

```
&APPC REGISTER ...  CONNECT=NOTIFY
```

Once this mode is set any pending connection requests, or any subsequent connection requests, are notified to the server by an event message being queued to the process's internal environment. This message is the same as that created by a transfer request from another process, and the server accepts or rejects these connections in the same manner as a transfer request. An &INTREAD statement can be used to receive the notification, informing the process of the conversation identifier that is pending connection. The server can choose to accept the connection request and begin processing the transaction. For example:

```
&APPC TRANSFER_ACCEPT ID=&id VARS=PIP*
```

In this case, it is possible for the server to obtain any PIP data present in the attach request. However, the server might choose to reject the new transaction. For example:

```
&APPC TRANSFER_REJECT ID=&id RETRY=YES
```

This is manifested in the remote allocation system as an allocation failure, with a reason of either:

```
TRANS_PGM_NOT_AVAIL_RETRY
```

or

```
TRANS_PGM_NOT_AVAIL_NO_RETRY
```

If the connection is accepted, the conversation is connected in receive state and is operated in the usual manner. This form is most useful when the server is to continue to operate in notification mode, by using the asynchronous form of receipt. For example:

```
&APPC RECEIVE_NOTIFY ID=CLIENTS
```

Rejection Mode

Servers can optionally mask off further connections by setting the connection mode to reject, as follows:

```
&APPC SET_SERVER_MODE CONNECT=REJECT RETRY=NO
```

In this case the RETRY option can be YES or NO, and subsequent connections are rejected as for a transfer reject. This can be useful where the server is terminating to indicate whether or not processing can be retried.

Transferring a Conversation

Management Services supports some implementation-specific &APPC verb options that allow an active conversation to be transferred from one NCL process to another. These are as follows:

```
&APPC TRANSFER_REQUEST
&APPC TRANSFER_ACCEPT
&APPC TRANSFER_REJECT
```

The TRANSFER_REQUEST option requires that a target NCL identifier be specified. This NCL process is then notified of the transfer request by a message queued to its internal environment. It can accept or reject the conversation transfer (as indicated by the syntax shown above) which completes the transfer request. After the transfer, the requesting procedure has no access to the conversation.

A transfer can only take place while the conversation is in send or receive state. Most likely this is immediately following the completion of an allocate or attach request. Using this technique it is possible to pass additional conversations to a process that is already handling other conversations.

13

The Program-to-Program Interface (PPI)

Program-to-Program Interface (PPI) provides a general-purpose facility for programs, written in any language, to exchange data. It also provides a facility for any program to forward a generic alert to Management Services.

This chapter discusses the following topics:

- Uses of PPI
- The CNMNETM Module
- Structure and Data Flow
- Interface Details
- The &PPI Verb

Uses of PPI

As PPI is available to any environment, not just NCL, PPI provides a simple, powerful technique to access Management Services from outside Management Services.

For example, an NCL process could provide a batch program with the ability to issue selected Management Services commands and return the results of the command to it.

PPI also provides an alternative method of communication between two NCL procedures, with no data loss if Management Services terminates. The data remains queued in the PPI server address space. The NCL procedures need not be active on the same Management Services.

No special authorization is required to use PPI, and it does not depend on having Management Services running.

The initial Management Services implementation is currently supported on MVS/SP, MVS/XA,MVS/ESA, MSP, MSP/AE, MSP/EX and VOS3/AS.

The PPI implementation can use either Cross-Memory Services or Service Request Block (SRB) scheduling, so that MSP can be supported.

The NCL &PPI verb provides access to the Management Services Program-to-Program Interface. This interface allows any programs, executing in the same MVS system, written in almost any programming language to freely exchange information.

The API provided by the Program-to-Program Interface is described in the IBM manual *NetView Application Programming Guide: Program to Program Interface*.

PPI services can be provided by the Management Services Subsystem Interface (SSI), or by the NetView Subsystem Interface.

The CNMNETM Module

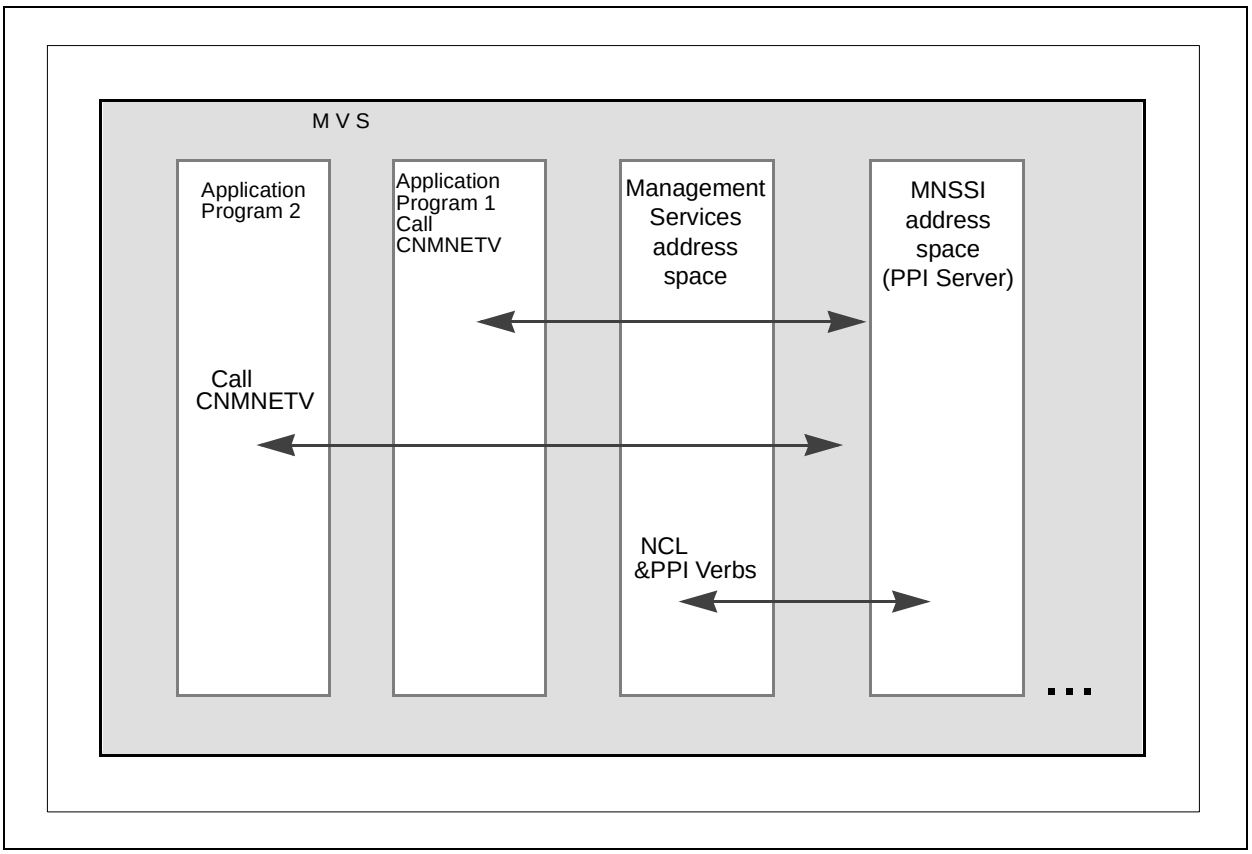
The *NetView Application Programming Guide: Program to Program Interface* discusses the CNMNETV module that you call when using the PPI API. A similar module, CNMNETM, with an alias of CNMNETV, is provided. This module can be loaded by application programs, or link-edited with them. It is fully re-entrant and can be placed in the PLPA, if so desired.

At this time, the IBM and Management Services versions of this module are incompatible with each other's implementation of the PPI. That is, the IBM CNMNETV module will not work in a Management Services PPI environment, and vice versa.

A subsystem is available for Management Services which runs as a separate job or started task. This allows PPI to stay active regardless of whether Management Services is active or not. A command/control interface between the subsystem and Management Services, allows control of the subsystem from any suitably authorized Management Services command environment.

Structure and Data Flow

Figure 13-1. PPI Structure and Data Flow



Programs issue calls to CNMNETM (CNMNETV), to send data to the nominated receiver program (see figure 13-1). The data is buffered in the SSI address space in queues associated with each receiver. When data is available for a given receiver, Management Services posts an ECB that the receiver can wait on, and the receiver can then call CNMNETM using a receiver function to obtain the next data buffer.

Note

Data is not directly moved from sender to receiver. The buffering allows the sender or receiver to run asynchronously.

Interface Details

A brief overview of the interface follows. (For more information on the Application Programming Interface (API). see the chapter titled *Program to Program Interface (PPI) Support* in the *Management Services User's Guide*.)

- Programs construct a request parameter block (RPB) that contains details about a call to PPI.
- A program then issues a CALL to CNMNETV (CNMNETM), passing the RPB as the only parameter (standard linkage, R1 pointing to the word containing the RPB address).
- Upon return, the RC field in the RPB contains the return code. Depending on the call and the return code, other data can be provided. See the &PPI verb description in the *Network Control Language Reference* for a table that shows the correlation between the &ZFDBK and &RETCODE system variables.

The following functions can be provided in the RPB function code field:

Function code 1—inquire about PPI status

This function code allows an application program to determine if PPI is available and active in the system. A return code of 10 indicates that it is available.

Function code 2—obtain receiver status

This function code allows an application program to determine the status of a named receiver program. Return codes indicate whether the receiver is defined or not, and whether it is presently active or not.

Function code 3—obtain ASCB and TCB addresses

This function code allows programs written in languages that do not support the ability to obtain ASCB or TCB addresses, the addresses of these control blocks. They are needed for other PPI calls.

Function code 4—define and initialize receiver

This function code allows a program to define itself as a receiver. The program provides a unique 1 to 8 character receiver name and queue limit. Following this call, the receiver is defined, and other programs can send data to it.

The receiver can optionally be defined as authorized. This prevents programs that are not APF authorized from sending data to it.

Function code 9—deactivate a receiver

This function code allows a defined receiver program to deactivate itself. The queue limit can be optionally altered. If a receiver program (Task) terminates without issuing this call, it is automatically deactivated.

Function code 12—send a generic alert

This function code allows any program to send a generic alert to Management Services. It is sent to the defined receiver NETVALET, which receives these generic alerts in NMVT format, and forwards them on to NEWS.

Function code 14—send a data buffer

This function code allows any program to send a data buffer to any defined receiver (active or inactive). The receiver will be notified if necessary.

Function code 22—receive a data buffer

The function code allows a receiver program to receive the next available data buffer. If no buffers are available, a return code informs the receiver.

Function code 24—wait on an ECB

This function code allows a program written in a language that does not support ECB waiting, to wait for data to arrive.

Function code 60—obtain a unique name

This function code, available only in the Management Services implementation of PPI, allows a program to obtain a unique 8 character ID. Although programs that only send data need not have a unique ID, all receivers must have a unique ID. This service is provided to allow a sender to obtain a unique ID so that it can register itself as a receiver for a two-way conversation.

The &PPI Verb

All PPI facilities can be accessed using the &PPI verb. The specific request is identified by a keyword immediately following the &PPI verb. These keywords generally correspond to the various functions described in the API.

The full set of &PPI requests is:

- &PPI ALERT
- &PPI DEACTIVATE
- &PPI DEFINE
- &PPI RECEIVE
- &PPI SEND
- &PPI STATUS

The &PPI verb, syntax, and examples of its use are fully described in the *Network Control Language Reference*.

Return Codes, System Variables, and User Variables

Following each execution of the &PPI verb, the &RETCODE and &ZFDBK system variables are set to reflect the success or otherwise of the request. &RETCODE contains a normalized return code. &ZFDBK contains the PPI return code, as documented in the chapter titled *Linking Programs* in the *Management Services User's Guide*.

See the table, *The Correlation Between the &ZFDBK and &RETCODE System Variables*, which is part of the &PPI verb description in the *Network Control Language Reference*. This table shows the returned values of each of these system variables.

Other system variables are:

&ZPPI
Indicates whether this system appears to support PPI or not.

&ZPPINAME
Contains the PPI receiver ID that this NCL process is registered as.

Some &PPI functions set specific NCL user variables:

&PPISENDERID
Set to the PPI ID of the sender of a received message.

&PPIDATALEN
Set to the actual received data length, in bytes.

Determining PPI or Receiver Status

The STATUS option of the &PPI verb allows an NCL process to determine the status of PPI itself (available or not), or the status of a PPI RECEIVER (by using the ID=*name* operand).

In either case, the process can examine the &RETCODE and &ZFDBK system variables after the request. If &RETCODE is 0, then PPI or the receiver is available or defined.

Defining the Process as a Registered PPI Receiver

By using the DEFINE option of the &PPI verb, an NCL process can register itself as a receiver. A 1- to 8-character name can be supplied, which must be unique (that is, not presently defined to PPI or currently inactive). If Management Services is providing PPI services, an alternative is to use the ID=* option, which causes PPI to provide a unique name. This option is useful when talking to globally named servers, as you need not worry about trying to find a unique name.

A process need not be defined to send data using the SEND and ALERT options. In this case, a sender ID of *#nclid* (7 characters, for example #001352) is used.

Sending a Generic Alert

One function of the PPI facility is to collect generic alerts and forward them to general CNM reporting (for example, NEWS). The &PPI ALERT verb allows any NCL process to send an alert to CNM. The alert must be formatted as an NMVT, including the NMVT header.

Sending Data to a Receiver

The &PPI SEND verb option allows any NCL process to send data to a nominated receiver. This receiver must be defined, but can be inactive (in which case data is queued unless the queue limit is reached).

Note

The receiver cannot be an NCL process at all, and can reside in another MVS address space.

The data to be sent can be a character string, HEX data that is packed before sending, or an MDO.

Receiving Data

An NCL process can receive data directed to its defined receiver ID using the RECEIVE option of the &PPI verb. That data can come from other NCL processes, including other SOLVE systems, or from other programs.

Standard parsing options, as on the other Management Services &*xxx*READ verbs, can be used. Alternatively, MDOs can be received.

The WAIT operand allows the procedure to indicate whether or not it will wait if no data is available, and, if no data is available, how long it will wait. Alternatively, the process can use WAIT=NOTIFY, to cause a message to be delivered to the dependent response queue when data arrives, thus allowing other work to be performed. When the notification arrives via &INTREAD, the process can reissue the &PPI RECEIVE.

Deactivating the Receiver ID

The &PPI DEACTIVATE option allows an NCL process to disconnect itself from a defined PPI receiver ID. Optionally, a queue limit can be specified, allowing data to be queued even though no receiver is present. The ID can be reactivated later, by this or any other NCL process.

If an NCL process that is defined to PPI terminates, an automatic deactivation occurs.

14

Synchronizing Access to Resources

NCL applications involving multiple users often need a mechanism for controlling access to the same data or resource, by different users or processes. This control is particularly important in NCL systems that use UDBs to hold application-related data that can be updated by some users, and concurrently read by others.

NCL provides this mechanism through the use of *locks* which allow or deny access to *resources*. The same mechanism is also used to synchronize activity between separate processes by providing a semaphore capability.

This chapter discusses the following topics:

- Understanding Resources and Resource Locks
- The Resource Name Hierarchy
- Resource Naming Conventions
- The &LOCK Verb
- Using Resources as Semaphores

Understanding Resources and Resource Locks

A resource, as the term is used in this chapter, is not a real entity such as a file or a variable. It is a *name*, consisting of a *primary name* and optionally a *minor name*, to which NCL procedures refer.

The purpose of the lock mechanism is to provide assurance that a specific operation can be performed, at a particular point within an NCL procedure's logic, without interference or damage by other NCL processes that might be attempting to use the same data at the same time.

A common example is the case where a process needs to update a record on a UDB. The process reads the target record, changes it and writes it back to the file, but it also needs to guarantee that no other process updates the same record at the same time. To achieve this guarantee, the process first obtains exclusive access to a resource that symbolizes the record update process.

Once it has obtained this exclusive access, the procedure is free to perform as much work on the target record as it needs, knowing that no other procedure can access the same record in the meantime because no other procedure would be able to gain access to the *resource lock*.

It is important to remember that it is not the *data* itself that is protected by the lock mechanism, only the *resource*. If controlled access to an item of data is required then all procedures that change that data must gain access to the lock before updating the data.

Resource Groups

To explain the concept of a *resource group*, take the example quoted above of a file record, in which a procedure uses resource locking to guarantee exclusive access to a file record for the purpose of updating it, without having to worry about any other process changing the record at the same time. In this example, the *resource* could be a name that represents the entire file; in other words, you could organize your procedure to have exclusive read/write access to a whole database.

Alternatively, and probably in preference, you would like other processes to continue to have access to the file as a whole, as long as they could not access the specific record that your process is updating. To achieve this, you would assign a resource *primary name* to represent the file (UDB) itself, and then decide on a naming convention that allows individual records within the file to be identified by a *minor name*.

For example, if you have a UDB containing NCP configuration data keyed by line name, you might assign a primary name of CONFIG to represent the UDB itself and use the line name as a minor name to represent each line record as a resource within the UDB. To retrieve information about a line, your procedure first obtains exclusive access to the appropriate line record *resource* by requesting exclusive control of the appropriate primary/minor name resource lock.

In this example, your procedure would execute the following &LOCK statement to gain exclusive permission to process the record on the UDB that contains information about line 23:

```
&LOCK TYPE=EXCL PNAME=CONFIG MNAME=LINE23
```

You could code the following statement if your naming convention for resource locks uses XYZ to identify the record on the database that describes the configuration of line 23:

```
&LOCK TYPE=EXCL PNAME=CONFIG MNAME=XYZ
```

In this example, the *primary name* (CONFIG) represents a resource group. The combination of the primary name and a minor name identifies a resource within the group.

Primary Names

The primary name is the part of the resource name that uniquely identifies the resource group. It is a 1- to 16-character string of your choice. Logically, the primary name represents the root of a (potential) two-level hierarchy, below which one or more dependent *minor names* can exist.

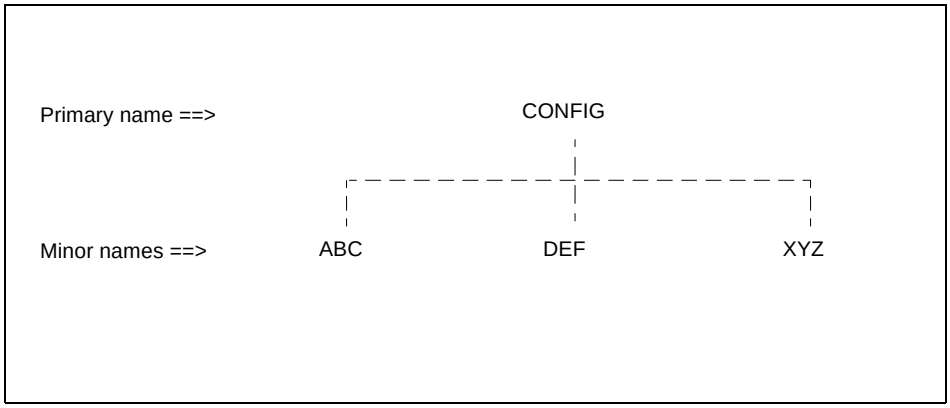
Minor Names

The minor name of a resource, if required, is a 1- to 256-character string, which qualifies the resource's primary name to identify a specific resource within a resource group. For example, the primary name CONFIG represents both a resource group and a specific resource. The resources CONFIG.ABC and CONFIG.XYZ represent specific resources within the CONFIG resource group.

The Resource Name Hierarchy

Specific resource names, that is, those that are explicitly defined by primary and minor name, are peers within their resource group. This is shown in Figure 14-1.

Figure 14-1. The Resource Name Hierarchy



Any process that obtains exclusive control of the resource CONFIG prevents any other process in the system from gaining access to any other resource below CONFIG in the hierarchy. Therefore no process can gain access to the CONFIG.XYZ resource, until the first process releases its exclusive control of the CONFIG resource lock.

Alternatively, a process that requests exclusive control of the CONFIG.ABC resource does not prevent any other process from accessing the CONFIG.XYZ resource.

Resource Naming Conventions

Since resources are only names, not real entities, the NCL procedures that use the locking mechanism to control and synchronize their access to different resources must agree on the resource names that they will use. They must also agree that a resource name means the same thing to all procedures. Once the resource names have been agreed, all procedures must use the &LOCK verb to obtain the resource lock to access resources, otherwise protection cannot be guaranteed.

It is very important that you define a naming convention for resource identification. Ideally your naming convention should apply to all SOLVE systems within your organization. Alternatively, naming conventions can apply within a given SOLVE system or within a specific NCL system.

The &LOCK Verb

The &LOCK verb allows you to obtain and release resource locks.

- &LOCK operates system-wide; it is not restricted to your NCL region alone.
- &LOCK is used to designate a particular resource lock that the procedure wants to control, and specifies the type of control that is required.

If a process has *exclusive* control of a resource lock, all other processes in the system are prevented from gaining access to the resource at the same time. Alternatively, you might want the process to gain *shared* access to the resource lock, which prevents any other procedure from being granted exclusive access. If required, you can alter the status of the lock from shared to exclusive, or from exclusive to shared, during processing.

A procedure can also use &LOCK to test whether any other procedure is holding a lock that it needs to access.

For a detailed description of the &LOCK verb and its use, see the *Network Control Language Reference* manual.

Waiting for Access to a Resource

If a procedure issues &LOCK to request access to a resource lock, but the required access cannot be granted immediately, the WAIT operand specifies whether the system will return control to the procedure immediately, or wait for the resource to become available.

If you specify WAIT=YES, your procedure will be suspended indefinitely until access to the required resource can be obtained. You should avoid this, unless you are certain that a deadlock condition with another process in the system will not result.

Rather than code WAIT=YES, it is recommended that you use the WAIT=*nnnn* option. This instructs the system to wait for the required resource for a certain number of seconds (as specified by the *nnnn* variable), rather than waiting indefinitely. If the lock is still not available after that time, the procedure resumes processing. This stops deadlock conditions occurring.

Altering the Status of a Resource Lock

If required, you can alter the resource lock status of a process during processing by using the ALTER=YES operand on the &LOCK verb. Altering the lock status from exclusive to shared is always possible, however there are some restrictions when altering the lock status from shared to exclusive.

Altering the Status from EXCL to SHR

During processing of the &LOCK request, any other lock requests that are waiting for shared access to the resource become valid for shared ownership. They are granted shared access to the resource immediately, causing the requesting procedures to resume execution.

Altering the Status from SHR to EXCL

The following conditions must be satisfied before a request to alter a resource lock status from shared to exclusive will be successful:

- No other procedures can have shared ownership of the resource
- If the resource is the primary resource, there must be no other minor resources (with shared or exclusive status) with the same primary name

If any other shared requests for the lock arrive before the status is altered to exclusive, these new shared requests are given precedence over the change to exclusive, and are granted shared ownership of the lock

If the request is successful the procedure owns the lock exclusively.

Using the WAIT Operand

As with normal shared and exclusive requests, the WAIT operand plays an important part in determining the success or failure of a request to alter the lock status. However, the waiting period is only significant for a request to change from shared to exclusive, as the request to change from exclusive to shared is always satisfied immediately.

When WAIT=NO is specified, the request fails unless it is satisfied immediately.

When WAIT=*nnnn* or WAIT=YES is specified, the requesting process can wait for the specified period of time for the change to be successful (this will happen when another process releases its lock).

If the status type on the &LOCK ALTER request is the same as the current status, the request is treated as a request to alter the lock *text* only—for example, the process holds a shared lock then issues a shared lock request with the ALTER=YES operand. This request is always successful.

Associating Text with a Resource Lock

In order to assist recognition, or to provide information to other processes that might need to interrogate the status of a particular lock, the &LOCK verb also allows you to specify text that is to be logically associated with your procedure's ownership of the lock. This text appears in the SHOW LOCKS command display, which is used to display the locks held by the different processes in the system. The text is also made available to a procedure issuing an &LOCK statement with the TYPE=TEST option.

For a complete description of the SHOW LOCKS command and its use, see the *Management Services Command Reference* manual.

Using Resources as Semaphores

Using &LOCK to synchronize the use of a resource between competing processes solves the problem of access to real resources. The same technique can be employed to signal between co-operating processes, in particular with the use of the TEST facility of &LOCK. In this case, the *resource* is not a real resource but a signal or *semaphore*.

Semaphores are a useful mechanism for synchronizing the processing of related procedures. Take an example of Procedure A that needs to suspend processing until Procedure B reaches a particular point in its processing.

Procedure A's logic could look like this:

```
.
.  -* process
.
&LOCK WAIT=YES TYPE=TEST PNAME=XYZ
```

At this point, Procedure A is suspended until another &LOCK request is made for the resource XYZ.

Procedure B in the meantime continues processing until it completes the function which it knows procedure A has to wait for. At this point, procedure B issues the same &LOCK request for the resource XYZ:

```
&LOCK WAIT=YES TYPE=TEST PNAME=XYZ
```

Both the &LOCK verbs complete with &RETCODE = 8, indicating that both procedures have reached the synchronization point. If Procedure B reaches the point first, its &LOCK would suspend it until Procedure A issued its &LOCK statement.

15

The NCL Debug Facility

This chapter discusses the following topics:

- Overview
- Security
- NCL Debug Facilities
- Controlling the Execution of NCL Processes

Overview

The NCL Debug facility is a powerful tool to assist in the debugging of NCL procedures.

The DEBUG command has various operands which provide the NCL developer with the ability to establish a debug session. This debug session can be targeted at one or more debug *scopes*. A debug scope can be any of the following:

- An NCL process
- All the processes within a particular user's window
- All the processes within a particular user's logon region
- All the processes for a particular user ID (that is, all regions for that user ID)

You can have one or more debug sessions active simultaneously, each being initiated from a different environment.

A debug session is associated with a particular region that is referred to as the *debugger*. This region is the only region from which debug commands will be accepted for the session. Any command environment can be a debugger. This includes OCS and any &INTCMD environment.

An NCL process that falls into the scope of a debugger is associated with that debugger's environment and is referred to as a *debugged* NCL process. An NCL process can only be debugged by one debugger at a time.

Once a debug session has been started, you can set break points in any procedure of any NCL process that has been attached to the debug session. The breakpoints can be set at particular points within the executed code, against the action of updating a variable, or against the execution of a nominated verb. The data of the target NCL process can be viewed and/or altered by the debugger. These breakpoints are completely external to the source and allow you to debug a process without having to modify the source code.

You can control execution of an NCL process by setting various breakpoints throughout the process. The DEBUG commands allow you to halt the execution of a process, resume execution (after a halt or breakpoint has been reached) or to step through a fixed number of process statements. You can also suspend the debug session and re-established it at a later time. Any breakpoints or suspended processes will be preserved in their current state.

The SHOW DEBUG command allows you to see all your current debug sessions and their associated scopes and, optionally, the debug sessions of other users.

For a full description of the syntax of the DEBUG and SHOW DEBUG commands, see the *Management Services Command Reference*.

Security

A set of security constraints is incorporated with this flexible debugging interface, to protect against intentional access to user's NCL processes that are running.

- Command authority level can be used to stop a user from debugging another user's procedures.
- NCL Debug is a feature of Management Services, and therefore can be *excluded* from a SOLVE system using an EXC= JCL parameter. (See the appendix titled *SOLVE JCL Parameters* in the *Management Services Implementation and Administration Guide* for details of using the EXC= parameter.)
- The DEBUG command can be replaced by an NCL procedure using the SYSPARMS CMDREPL operand. (This is described in the Appendix D, *SYSPARMS* in the *Management Services Implementation and Administration Guide*.)
- The DEBUG DISPLAY command can be used to display the contents of variables but it does not display the &USERPW variable.

NCL Debug Facilities

NCL Debug provides the following facilities:

- Allows observation of the execution of an NCL procedure from an external source, that is, another environment, another user region, window, or NCL environment.
- Eliminates the need for code changes to debug a procedure. For example, there is no need to add &CONTROL or &WRITE statements.
- Provides comprehensive control over the NCL procedure as it is being executed, supporting statement stepping, alteration of variable contents and attributes, and so on.
- Allows specification of criteria for debugging, before the target NCL begins execution.

Using the NCL Debug Facility

The NCL Debug facility is made up of a set of commands that allow you to do the following:

- Start and stop an NCL debug session
- Control the execution of NCL processes
- Display and modify the contents of NCL variables, independent of the nesting level of the process
- List the procedure and subroutine nesting levels
- Display the source code that is being executed (not the source currently on disk, that is, without comments or indentations)
- Receive NCL trace output at another region, thus being able to view the trace output concurrently as the debugged process executes

Starting and Stopping an NCL Debug Session

An NCL debug session is started in an environment by issuing the first DEBUG START command. The environment and scope of the session are defined using the operands of the DEBUG START command. Subsequent DEBUG START commands add additional scopes to the debug environment already established.

When the DEBUG START is processed, any NCL processes that fall within the specified scope, and which are not already debugged, will be attached to the debugger's session. If an NCL process starts, and its environment is being debugged, or it falls within the scope of a debug session, it will be attached to that debug session.

Once an environment has established a debug session, other debug commands can be issued to control the NCL processes that have been attached. The commands to manipulate these processes must be issued from the same environment that issued the DEBUG START. The process has no awareness of being debugged.

Once debugging is complete, the debug session can be terminated using the DEBUG STOP command. There are three options for the DEBUG STOP command that affect the actual processes that have been attached to the debugger.

- The CONTINUE option tells debug to remove all breakpoints from the processes and resume any that are suspended, leaving them to continue as if no debugging had taken place.
- The FLUSH option causes all attached processes to be flushed.
- The SUSPEND option leaves the debug environment intact and disconnects it from the debugger's environment (the debug session is suspended).

The CONTINUE and FLUSH options both result in the debug session being deleted. All breakpoints and scopes are cleared and any new processes that start, and fall within the scope, will not be attached.

The SUSPEND option leaves the debug environment intact. Breakpoints and scopes remain in effect. If a process starts within a scope, that process will be attached to the debug environment and any breakpoints that are pertinent will be applied. The debug session will be flagged as suspended, and a debug ID will be assigned to it. When the debug session needs to be reconnected, a DEBUG START command can be issued specifying the DEBUGID that was assigned to the suspended debug session.

A debug session can be reconnected to another environment of the same user ID. Conversely, a session can be reconnected from another environment of the same user ID. During the time that the session was suspended, new processes can be attached from the debug session. If an attached process encounters a breakpoint, it is suspended, and remains suspended until the debug session is reconnected and a command issued to resume processing for the process. The same applies for a process that was suspended at the time the debug session was suspended.

The following sequence of commands shows how to establish and stop a debug session:

```
DEBUG START WINDOW=2 -* debug any procedure in
                     -* window 2

DEBUG START USER=NM1BSYS PROCEDURE=MYBPROC
                     -* debug background procedure
                     -* in BSYS region

-* other debug commands control the process in the two scopes

DEBUG STOP TYPE=FLUSH-* terminate the debug
                     -* session and flush the
                     -* procedures
```

Controlling the Execution of NCL Processes

Once a debug session has been established, the debugger can issue various commands to control the execution of the processes that are subsequently attached to the debug session. The commands that can be used have the effect of either suspending a process, resuming the execution of a process, or both.

The following commands can be used to suspend an NCL process while it is executing:

DEBUG HOLD
: This command flags the process for immediate suspension, before the next statement is executed.

DEBUG STEP
: This command indicates to debug that the process is to be suspended after the specified number of statements have been executed. The NEXT= operand can be used to indicate how many statements of the process to allow before the process is suspended.

Both commands have the effect of putting the NCL process into a wait state. The SHOW NCL command indicates that the process has been suspended.

Another way for a process to become suspended is indirectly, using the DEBUG BREAKPOINT command. The DEBUG BREAKPOINT command defines a condition that must be satisfied before the process is suspended. The following conditions can be specified:

- Statement
- Verb
- Variable
- Procedure ENTRY
- Procedure EXIT

Statement Breakpoints

A *statement* breakpoint can be used when the process is to be suspended if the statement identified by the STMT= operand of the DEBUG BREAKPOINT command is about to be executed. The statement is identified using the line number found in columns 73 to 80 of the source, or a relative number if the source is unnumbered.

Verb Breakpoints

A *verb* breakpoint can be used when the process is to be suspended if the statement to be executed contains the verb identified by the VERB= operand of the DEBUG BREAKPOINT command. The process is suspended immediately *before* the verb is executed.

Variable Breakpoints

A *variable* breakpoint can be used when the process is to be suspended if a variable, identified by the VARS= operand of the DEBUG BREAKPOINT command, is updated. The process is suspended immediately *after* the statement that caused the variable to be updated. Optionally, the command can specify the DATA= operand to limit the breakpoint to updates of the variables with a particular value.

Procedure ENTRY Breakpoints

A *procedure ENTRY* breakpoint can be used to suspend execution of a procedure *before* the first statement of the procedure is executed.

Procedure EXIT Breakpoints

A *procedure EXIT* breakpoint can be used to suspend execution of a procedure *after* the last statement of the procedure has been executed. This includes any normal termination statements which end the procedure (for example, &END or &RETURN).

Using the BREAKPOINT Command

Breakpoint definitions are associated with the debug session, and are applied to procedures both when they are executed and when the breakpoint is defined. Thus, breakpoints can be defined before any processes have been attached to the debug session. When a process issues an EXEC command, the procedure being executed has any relevant breakpoints applied.

Statement breakpoints require a privately loaded copy of the procedure. (This is not necessary for the other types of breakpoint.) If the procedure is the target of a statement breakpoint, the debugger will automatically request that a procedure be privately loaded. Otherwise, normal procedure loading will remain in effect, as controlled by the SYSPARMS NCLTEST or PROFILE command.

Once breakpoints have been established, any NCL process that satisfies the conditions of a breakpoint is suspended. Breakpoints can be listed and cleared using the DEBUG LIST BREAKPOINTS command and the DEBUG CLEAR respectively. These commands are useful for controlling processes that have already started.

If you need to stop a process on the first statement, use the DEBUG SET NEWHOLD=YES profile command. Once set, it indicates to debug that all processes that are attached to this debug session are to be immediately suspended before the execution of the first statement.

When a process is suspended, other debug commands can be used to display variables and the source code.

To resume the execution of the process after all the required information has been displayed, use the following commands:

DEBUG RESUME

RESUME flags the process identified as ready to continue execution. The process is made eligible for execution and the statement step counter, set by the DEBUG STEP command, is reset.

DEBUG STEP

The process is resumed in the same way as with the DEBUG RESUME command and the statement step counter is set to the value specified on the NEXT= operand. This will result in the process again being suspended after the specified number of statements have executed.

Sample Debug Session

The following sequence of commands shows how to set breakpoints and control the execution of a process:

```
-* The debug session has already been established

DEBUG SET NEWHOLD=YES    -* Ensure the procedure is stopped
                         -* on the first statement

DEBUG BREAKPOINT PROCEDURE=MYPROC STMT=1250000
                         -* Suspend the procedure if it
                         -* tries to execute statement 1250000

DEBUG BREAKPOINT PROCEDURE=MYPROC VERB=APPC
                         -* Suspend background procedure on
                         -* first APPC verb

DEBUG BREAKPOINT PROCEDURE=MYPROC VARS=WKTRANID DATA=CH22
                         -* Suspend the procedure when it sets
                         -* WKTRANID to the data that causes
                         -* the error.  The procedure MYPROC is
                         -* started in window 2 and is
                         -* immediately suspended

DEBUG STEP NEXT=10       -* Execute the first 10 statements
-* Using the display commands the procedure is verified as
-* to the current state of its variables
DEBUG RESUME             -* Let process continue
                         -* until first breakpoint
-* The procedure is suspended on the APPC verb. using the
-* Using the display commands, the procedure is verified as to
-* the current state of its variables.  They are set correctly.

DEBUG LIST BREAKPOINTS   -* List all current breakpoints

DEBUG CLEAR BREAKPOINT=2-* Clear the breakpoint on the
                         -* APPC verb
DEBUG RESUME             -* Let the process continue until
                         -* next breakpoint

-* Process continues
```

Displaying and Modifying the Contents of NCL Variables

Once the procedure has reached a particular point in its processing and it has been suspended, the DEBUG DISPLAY and DEBUG MODIFY commands can be used to display and modify the contents of variables and MDOs. The entire contents of the variable or MDO are displayed.

The modify command can be used to alter the attributes and contents of variables and MDOs. Multiple variables can be altered by specifying the operands of the modify in a similar fashion to using the &ASSIGN verb, however the new value can only be specified using the DATA= operand.

See the section *Example Session* later in this chapter for examples of these commands.

Listing Procedure and Subroutine Nesting Levels

The DEBUG TRACE command is used to list all the procedures. Subroutine nesting levels can also be obtained. The listing shows a general display of the identified process, and then a detailed list of each procedure and the subroutines that the procedure has entered.

See the section *Example Session* later in this chapter for examples of this command.

Displaying the Executed Source

The DEBUG SOURCE command allows you to list the statements of a procedure, as they are stored in memory. The listing represents what the system will actually execute, as opposed to what is currently on disk. This can be useful in verifying that the procedure being debugged is the correct version.

See the section *Example Session* later in this chapter for examples of this command.

Receiving NCL Trace Output

When debugging full screen procedures, the output from the NCLTRACE facility is not available for viewing until the process has completed, or the window has been released by the process. This can be undesirable if the process has been suspended after a panel has been sent and the process still owns the window. The trace output will not be seen until the process releases ownership of the window.

Using the DEBUG SET NCLTRACE=YES profile command, the trace output from a process being debugged will be delivered to the debugger's environment.

See the next section for examples of this command.

Example Session

The following sequence of commands shows how a debug session could proceed:

```
DEBUG START WINDOW=2           -* Debug any procedures in
                               -* window 2
DEBUG START USER=NM1BSYS PROCEDURE=MYPROC
                               -* Debug background procedure
                               -* MYPROC in BSYS region

DEBUG SET NEWHOLD=YES          -* Ensure procedure is
                               -* stopped on 1st statement

DEBUG BREAKPOINT PROCEDURE=MYPROC STMT=1250000
                               -* Suspend the procedure if it
                               -* tries to execute statement
                               -* number 1250000

DEBUG BREAKPOINT PROCEDURE=MYPROC VERB=APPC
                               -* Suspend background
                               -* procedure on the first
                               -* APPC verb

DEBUG BREAKPOINT PROCEDURE=MYPROC VARS=WKTRANID DATA=CH22
                               -* Suspend the procedure when
                               -* is sets WKTRANID to the
                               -* data that causes the error

-* The procedure MYPROC is started in window 2 and is
-* immediately suspended


DEBUG STEP NEXT=10             -* Execute the first 10
                               -* statements

DEBUG DISPLAY VARS=WK* GENERIC
                               -* Verify that the work
                               -* variables have been set
                               -* correctly

DEBUG RESUME                   -* Let the process continue
                               -* until the first breakpoint

-* The procedure is suspended on the APPC verb

DEBUG DISPLAY VARS=WKTRANID    -* The variable has the
                               -* correct value - continue


DEBUG LIST BREAKPOINTS         -* List the current
                               -* breakpoints

DEBUG CLEAR BREAKPOINT=2       -* Clear the breakpoint on
```

```
                                -* the APPC verb

DEBUG RESUME                    -* Let the process continue
                                -* until the next breakpoint
-* The procedure is suspended after updating WKTRANID to
-* CH22

DEBUG DISPLAY VARS=WK* GENERIC -* The display indicates that
                                -* WKTRANPREF was incorrect

DEBUG MODIFY VARS=WKTRANPREF DATA=ZCH
DEBUG MODIFY VARS=WKTRANID DATA=ZCH22
                                -* Fix the variables

DEBUG CLEAR                     -* Clear all the breakpoints

DEBUG BREAKPOINT VARS=WKTRANPREF
                                -* Stop the procedure when
                                -* WKTRANPREF is updated

DEBUG RESUME                    -* Let the process continue
                                -* until the next breakpoint

-* The procedure is suspended after updating WKTRANPREF

DEBUG TRACE                     -* Display the nesting
                                -* levels.

-* The display indicates that a subroutine was called at
-* the wrong place

DEBUG MODIFY VARS=WKAPPPPLCTR DATA=2
                                -* Change the loop counter to
                                -* go through the process again

NCLTRACE ON ID=123              -* Set tracing on

DEBUG SET NCLTRACE=YES          -* Have the trace output sent
                                -* here

DEBUG STEP NEXT=80              -* Check the results

-* After seeing the trace, it is clear that the variable WKHSGT
-* is incorrectly set.  The source is amended.
-* The bug had been found, so stop the debug session and flush
-* the process.

DEBUG STOP TYPE=FLUSH           -* Terminate the debug
                                -* session and flush the
                                -* procedures
```

16

NDB Concepts

This chapter describes NetMaster Databases (NDBs). It covers the differences between NDBs and User Databases (UDBs), the structure of an NDB, and the uses to which an NDB can be put.

Familiarity with standard UDBs is assumed, including the use of the &FILE verbs, and the UDBCTL command.

This chapter discusses the following topics:

- What is an NDB?
- NDB Structure
- Record ID (RID)
- NDB Data Formats
- Null Values and Null Fields
- NDB Transaction Management: Database Protection
- NDB Journaling

What is an NDB?

Management Services provides an enhanced database manipulation facility for NCL procedures. This facility allows data to be stored, to be retrieved or updated later, in a formatted database known as a NetMaster Database or NDB. The NDB format is more powerful than standard VSAM datasets.

Note

The term, database manager, used in this manual refers to the assembler code that controls NDBs. There is a database manager for each active NDB.

An NDB is a formatted VSAM Key Sequenced Dataset (KSDS). It should be accessed only by using the &NDB verbs. An NDB supports the following:

- Multiple keys, without any VSAM alternate indices
- Logical record size not limited by the defined VSAM record size
- Data access by named field, not relative field position in a record (as in a UDB)
- Transaction integrity, guaranteeing a non-corruptible file
- Multiple users, without VSAM string limitations
- Different data types, including character, numeric, floating point, hexadecimal, and date format data. The Database Manager prevents data that is not in the defined format and invalid data, from being stored (for example, the value ABC could not be stored in a field defined as numeric).
- An extremely powerful search capability, that makes full use of keys wherever possible, *but* does not require *any* keying of the search arguments.
- *Null field* support, which allows multiple record types to co-exist in a single NDB. This is defined in the section titled *Null Values and Null Fields*, on page 16-8.
- Forward recovery facilities, minimizing the risk of data loss. This is explained in the section titled *NDB Journaling*, on page 16-11.

Working with NDBs

You can use NDB commands to start, stop, reset, and lock (for example to prevent access while running a backup), an NDB. See the *Management Services Command Reference*.

You can also use NCL verbs (see the chapter, *Using &NDB Verbs*) to insert or delete field definitions, add, delete, update and retrieve records, and to search an NDB for all records that match a supplied set of criteria. These verbs also allow access to the database and field level definitions, making it easy to provide utility procedures that need only be told the name of the NDB to be accessed.

Uses of NDBs

An NDB can be used in any application where a flexible data storage mechanism is required. Applications that have complex retrieval needs are especially suited to NDBs. The ease of access of data makes such things as selection list scrolling very easy to perform.

NDBs are not suited to applications that have a large amount of *high-speed* record addition, update, or deletion.

Differences Between NDBs and UDBs

Both NDBs and UDBs are always VSAM datasets. An NDB is always a VSAM KSDS, whereas a UDB can be either a KSDS or an ESDS (a non-keyed VSAM dataset). The differences between a UDB and an NDB are shown in Table 16-1.

The UDBCTL command is used to physically open a VSAM dataset for access by the SOLVE Virtual File Services (VFS). Thus, although an NDB should not be used as a UDB, the UDBCTL command is still used to open and close it.

Warning

An NDB must not be accessed by the &FILE verbs, as if it were a UDB. The NDB can be corrupted if any access is done in this way.

Table 16-1. The Differences Between a UDB and an NDB

UDB	NDB
Can be a VSAM KSDS or ESDS.	Always a VSAM KSDS.
If KSDS, key length can be from 1 to 255.	Key length must be from 16 to 255.
VSAM maximum data length can be any valid value.	VSAM maximum data length has a minimum value restriction
Data is built/accessed by relative position in a record.	Data is accessed by field name.
Record length is limited by either VSAM record length, or the maximum NCL statement length after substitution.	Record length is independent of the VSAM record length or NCL statement length. Logically limited by the restriction of 32K named fields.
Only one, unique, sequence key is provided, unless VSAM alternate indices are used.	There is no requirement for any key. There can be any number of keys, and they need not be unique. There can, optionally, be a unique sequence key.
Multiple record structures to support multiple keying etc. Have no update integrity. A system failure can logically or physically corrupt the database.	Internal update journaling guarantees integrity of the multiple VSAM record structures used to support the NDB features.
Concurrent access can result in VSAM string waits or lockouts.	Concurrent access has no restrictions.
Complex searching is slow, as the logic must be implemented in NCL.	Complex searching runs at assembler speed. Keys are used whenever possible.
Multiple positioning (for sequential retrieval) by one NCL procedure is not possible.	An NCL procedure can have any number of simultaneous positions in an NDB.
UDBSs are accessed using the &FILEOPEN verb. The first &FILE OPEN for a given UDB performs a logical open for the NCL process. Since the &FILE GET, and similar, verbs have no way of specifying the UDB they refer to, the last executed &FILE OPEN statement sets the UDB for these statements. Thus, &FILE OPEN has a double meaning.	NDBs are accessed using the &NDBOPEN verb. Each NDB to be accessed must be opened by an &NDBOPEN statement. The other &NDB*xxx* verbs allow specification of the database name and there is no need for repeated &FILE OPEN-like use of the &NDBOPEN verb.
Data in a UDB has no associated type. No validity checking is performed on the stored data.	Data in an NDB has a type. Invalid data (for example, non-numeric for a numeric field) cannot be stored.

When an NDB is active, VFS prevents any access to it as a UDB. For example, if an &FILEID statement refers to an active NDB, it causes the NCL procedure to terminate with an error message.

Similarly, if an NDB is being accessed as a UDB, it cannot be accessed by NDB commands or &NDB statements.

NDB Structure

The internal structure of an NDB is described below. The records are described in ascending VSAM key sequence.

Control Record

The first record (key is always all binary 0) in an NDB is a control record. It identifies this dataset as an NDB, and contains other control information (for example, the number of defined fields and the number of records). This record is inserted when the NDB CREATE command formats an NDB. It is updated by other NDB commands and NCL statements. It also contains the SOLVE Domain ID and NDB name. These are used to prevent concurrent update access by more than one SOLVE system.

Journal Control Record

This record, also inserted when the NDB is formatted, is used to manage the NDB transaction journal. See the section titled *NDB Transaction Management: Database Protection* on page 16-10 for more details.

Journal Data Records

These records are used to journal update activity, to allow update retry after a system failure. There are *n* of these records, the number being determined either by the value of the NDBLOGSZ SYSPARM when the NDB CREATE command was issued for the NDB, or by the LOGSIZE parameter on the NDB CREATE command.

Field Definition Records

These records, one for each field defined on the database, contain information about the fields (for example, the field name, data format, and key options).

Key Statistics Records

These records, one for each field, contain key statistics information collected during an NDB ALTER BLDX or an NDB START KEYSTATS run.

Key Records

For fields defined as keyed (except a sequence key), these records contain the key values, and, for each key value, a list of the record IDs of records containing that value.

RID-Sequence Key Records

For NDBs defined with a sequence key, these records act as a link record, keyed by record ID, and contain the sequence key value of the record.

Data Records

These records hold the actual data for each record stored in the NDB. They are keyed by record ID, or by sequence key value (if a sequence key is defined for this NDB). If a given logical record has more data than will fit into a single VSAM record, it is automatically spanned across multiple records.

Record ID (RID)

The term, Record ID (RID), is used to describe the unique identifier of a logical record in an NDB.

The RID is assigned by the Database Manager, when a record is added to an NDB. The RID is a number, from 1 to 1 billion (actually, 2**30 - 1), that uniquely identifies this record in the NDB. All internal access to a record is made by providing the RID. The RID assigned to a record never changes, and (currently) is not reassigned to another record when a record is deleted.

Note

The RID cannot be assigned by the user. The assigned value is made available to the NCL process that inserts a record via the &NDBRID system variable.

The RID is necessary, as an NDB does not need to have a sequence key (or any key, for that matter), and a way of uniquely identifying a record is always required.

Whenever a record is retrieved by an &NDBGET NCL statement, the RID of that record is returned in &NDBRID. The record could have been retrieved by a key, or from an &NDBSCAN result list, but the RID is always made available, so that a following &NDBUPD or &NDBDEL statement can refer to the correct record.

NDB Data Formats

An NDB supports the following data formats:

CHAR

Data is provided, and stored, as a character string. NCL restricts character data to printable characters. The maximum length of a character field is 255 characters if not keyed, or (VSAM keylength - 8) if keyed. Character fields collate (for keying, or sorting in a scan) on ascending EBCDIC value. Trailing blanks are not significant and are removed from stored data. The option to automatically make data upper case is available. Alternatively, data can be stored as lower case but searched as if it were upper case.

NUMERIC

Data is provided as an optionally signed number, from -2,147,483,648 to +2,147,483,647. Numeric fields collate on ascending *binary* value (-10 before -5 before 0 before +5 before +10). The minimum VSAM key length for an NDB guarantees that numeric data can always be keyed.

HEX

Data is provided as an even number of hexadecimal characters (0-9, A-F, or a-f). Trailing blanks are eliminated from the value. Trailing zeros *are* significant, and are stored. The maximum length of the *character* representation of a HEX field is 254, giving a maximum binary length of 127. If keyed, the maximum *character* representation length is ((VSAM keylength - 9) * 2). A null-valued HEX field can be represented by a character value of one blank. HEX fields collate on ascending binary value, with values that are equal except for the number of trailing zeros collating on increasing length.

DATE

Data is provided in one of several formats, controlled by the user and/or system language code, and the current &NDBCTL DATEFMT setting. Basically, the provided value is in YYMMDD format. The data is stored internally as packed digits in the form YYMMDD. Date fields collate on ascending date value. The minimum VSAM key length for an NDB guarantees that date data can always be keyed.

Note

As the DATE format does not include a century, we recommend use of the CDATE format instead.

CDATE

Data is provided in one of several formats, controlled by the user and/or system language code, and the current &NDBCTL DATEFMT setting. The data is stored internally as a 3-byte binary number, being the numbers of days from 1/1/0001.

TIME

Data is provided in HHMMSS.TTTTTT format (the decimal point and fraction can be truncated or omitted). The data is stored internally as a 5-byte binary number, being the number of microseconds since midnight.

TIMESTAMP

Data is provided in YYYYYMMDDHHMMSS.TTTTTT format. The data is stored internally as a concatenation of a 3-byte CDATE and 5-byte TIME.

FLOAT

Data is provided as a floating point number. It is stored internally in IBM 8-byte normalized floating point format. The numbers are stored to 15 significant digits and with an exponent of +70. Floating point fields are collated on ascending numeric value.

Null Values and Null Fields

One of the most powerful features of NDBs is the use of *null fields,* an understanding of which is essential to the effective use of an NDB.

An NDB is a *field-oriented* database. Data is always accessed by field name. Other databases may have the concept of a *record* or *group,* being a collection of fields, that can be accessed by the name of the record or group. An NDB *record* is the actual, complete record, as logically accessed by RID.

At first, then, it might seem that an NDB can contain only one *record-type,* where *record-type* corresponds to a supplier, an order, or a customer record, for example. This is not the case. In fact, an NDB can contain almost any number of *logical record types*, each of which can be accessed separately. To achieve this, an NDB uses the concept of the *null field*. A null field is simply a defined field that is *not present* in a record.

The number of defined fields in the control record includes the number of null fields and each null field has a field definition record. However, the field is not physically present in the record. This is clearly not the same as a field that is *present, but contains a null value* (for example, blank).

Even if the field is defined as keyed, a null field is *never* keyed. Thus, any attempt to access by using keys will *never* retrieve a record that has that field *null.*

For any data format, a field can have one of the following logical values:

- Not present (that is, null field).
- Present, but *null-valued*. For a character field, this is defined as all blank. For a numeric or floating point field, this is defined as 0, for a hex field, this is defined as all blank (stored as present with a length of 0), and for a date field, this is defined as YYMMDD = 000000 (the only invalid date value allowed).

- Present, with a value other than the null value described above.

In NCL terms, the following statements illustrate the three states of a field:

&VALUE =	Sets &VALUE to not-present (the actual variable is deleted from the NCL procedure's variable pool).
&VALUE = &SETBLNK 1	Sets &VALUE to present, with the value of one blank.
&VALUE = *value*	Sets &VALUE to present, with the value of *value*.

Rules for Null Fields

There are several rules regarding null fields:

- A null field is never keyed. Thus, access using keys for that field name will never retrieve a record with that field a null field.
- A null field *never* matches anything, not even another null field. For example, none of the following &NDBSCAN statements will retrieve the entire database, if field NAME is null in some records.

```
&NDBSCAN dbname     FIELD NAME EQ VALUE 'FRED' OR +
                    FIELD NAME NE VALUE 'FRED'
&NDBSCAN dbname     FIELD NAME EQ FIELD NAME
```

  In the second example, comparing a field to itself in a record, it might seem that some records should be selected. The rule about null fields is still honored in this case.

- You cannot retrieve a record with null fields. Retrieving a record with null fields causes the NCL variables receiving the requested null fields to be deleted (that is, set to null).
- A null field can be indicated on an &NDBADD statement by omitting the *fieldname = fieldvalue* clause for that field, or, on an &NDBADD *or* an &NDBUPD statement, by using a currently undefined (that is, null) NCL variable as the *fieldvalue*, for example, FIELDX = &NULL, or by using the syntax FIELDX NULL.
- The only way that records containing a particular null field can be selected in an &NDBSCAN is to use the PRESENT and ABSENT or IS [NOT] NULL operators. These operators allow records to be selected that contain the nominated field (PRESENT) or records that *do not* contain the nominated field (ABSENT).

Using null fields, the previous example of a database containing supplier, order, and customer records can be built by inserting only supplier fields for supplier records, order fields for order records, and customer fields for customer records. The records are now *disjoint*. A retrieval by CUSTNO, for example, would only retrieve records containing the CUSTNO field, and so on for supplier and order.

When defining fields in an NDB, a field can be made mandatory by specifying NULLFIELD=NO.

NDB Transaction Management: Database Protection

An NDB is protected against system failures that might occur when an update to the VSAM dataset is in progress. Database record locking prevents data corruption caused by multiple users accessing the same record simultaneously.

If it is possible for more than one user to access an NDB record at once, use the &LOCK verb to ensure exclusive access to the record while it is being accessed. That is, perform an &LOCK on a record before any operation that accesses that record proceeds. If another user subsequently accesses the record, any modifications made by the second user do not proceed until the first accessing procedure concludes and the record ceases to be locked. See the *Network Control Language Reference* for a complete description of the &LOCK verb.

Protection against system failures is achieved as follows:

- A preformatted journal area is built when the NDB is created. This area consists of a journal control record, and *n* journal records.
- When an operation that involves updating the NDB starts, the journal area is used to record the updates, but they are not actually performed.
- When the updates are complete, the journaled updates are used to *physically* update the dataset. Before this starts, a flag is set in the journal control record, indicating an *update apply* procedure is in progress, and the journal control record, *and* all journal data records are force-written to the VSAM dataset.
- If the *update apply* completes successfully, then buffers are flushed, the journal control record flag is reset, and the journal control record is rewritten.
- If the *update apply* is interrupted by a system failure, then, when the NDB is next activated, the journal control record flag indicates that an *update apply* was in progress at the time of the failure. The entire *update apply* is redone (ignoring errors due to duplicate or already deleted records). Thus database integrity is assured.
- Physical errors on the VSAM dataset (for example, out of space) are handled in the same way. Once the dataset has been copied to a larger version, the reapply works in the same way.

NDB Journaling

In addition to the preformatted journal area kept within an NDB, an external journal can be created. This journal allows:

- Continuous availability of an NDB which cannot regularly be stopped for backups
- Recoverability of an NDB to the time of last update, even in the event of physical dataset failure

To enable journaling on an NDB, specify the JOURNAL operand on the NDB START command. This causes the after images of all NDB record updates to be written to the journal.

Note

Before images are not kept; therefore dataset *backout* is not possible.

Continuous Availability

If your NDB cannot be stopped for backup because of availability requirements, you can use the journal to keep a current backup copy. A backup copy is created once and the journal is applied to it each time the journal is swapped.

Note

A journal swaps if a JOURNAL SWAP command is issued, or if the current journal runs out of space.

Use the batch forward recovery utility (UTIL0010) to apply the journal. The name of the journal to be applied can be determined using the &ZJRNALT system variable. The sample batch forward recovery JCL ($NDUT010) is available in the distribution library and must be tailored using the installation dataset names.

Whenever a journal swap occurs, an NCL procedure is started in the BSYS environment to assist in the automation of forward recovery. The name of this procedure is specified using the SYSPARMS JRNLPROC command (default is $NDJPROC). Use this procedure to submit your NDB forward recovery JCL.

See *Using the NDB Journal* in Chapter 17, *Netmaster Database (NDB) Administration*, of this manual for more detail.

NDB Recovery

If you intend to continue taking regular backups of your NDBs, then you can choose to apply journals only in the event of physical loss of an NDB. When taking regular backups of your NDB you only need to backup your journal datasets whenever a journal swap occurs. The NDB can then be restored from a backup, applying all journals since backup in sequence.

17

Netmaster Database (NDB) Administration

This chapter describes the tasks you use to create, access, back up, delete, monitor, and control NDBs.

Familiarity with standard User Databases (UDBs) is assumed, including the use of the &FILE*xxx* verbs, and the UDBCTL command. A knowledge of VSAM, the IDCAMS utility program, and JCL (for the relevant operating system) is also assumed.

This chapter discusses the following topics:

- Creating an NDB
- Deleting an NDB
- Altering Field Definitions in an NDB
- Backing Up an NDB
- Restoring an NDB
- Monitoring NDB Activity
- Monitoring NDB Performance
- Multiple System Access to an NDB
- Using the NDB Journal

Creating an NDB

Perform the following tasks to create an NDB. Read the complete description of each task carefully before performing the task.

Task 1—Define VSAM Dataset

Use IDCAMS to define a VSAM KSDS. The following parameters are required:

INDEXED	Indicates a KSDS
KEYS (*len 0*)	keylength (see below)
RECORDSIZE (*avg max*)	Indicates average and maximum record size. See below.

Other parameters, such as SPEED, REPLICATE, IMBED, can be used at your discretion.

Do not use SPANNED. Logical records longer than the VSAM data record length are handled automatically.

The REUSE parameter can be used, but, for NDBs used in a production environment, it is not recommended, as omission prevents the accidental emptying of an NDB by use of the UDBCTL OPEN RESET command.

Task 2—Calculate Key Length

Calculate the key length to use for the NDB as follows:

L1 =	Longest desired length for any keyed CHAR field
L2 =	Longest desired length for any keyed HEX field (as displayable characters)
L3 =	Maximum of (L1 + 8), (L2 / 2 + 9), and 16
KL =	Minimum of L3, and 255.

Task 3—Calculate Record Length

Calculate the average and maximum record lengths as follows:

Average Record Length

- Greater than the VSAM key length
- The approximate size of a stored data record. A stored data record needs:
  - KL (from above) +10 +
  - 3 times number of *present* fields +
  - 4 times number of *present* numeric fields +
  - 3 times number of *present* date or *cdate* fields +
  - 8 times number of *present* floating point or timestamp fields +
  - 5 times number of *present* time fields +
  - total number of characters in present hex fields / 2
  - number of characters in *present* character fields (less trailing blanks) +

Note

A data record can span multiple VSAM records. If there are some records in your design that can be very large, but most are short, calculate the sizes using most common record structure.

Maximum Record Length

- A maximum of 274 + 2 times the VSAM key length
- Average record length (calculated above) + VSAM key length + 6

VSAM requires the maximum record length to be less than the data CI size less 7. Use a figure that leaves minimum wastage in a CI.

The only VSAM records in an NDB that have the maximum VSAM record length are the NDB transaction data records. These are inserted during processing of the NDB CREATE command.

A *rule of thumb* VSAM definition is shown below. This example of a definition is a good starting point for your own definitions.

```
DELETE clustername CL
DEFINE  CLUSTER(NAME(clustername)   -
        VOL(volume)                 -
        INDEXED-
        SHAREOPTION(2 3) -
        NOREPLICATE IMBED)-
        DATA (NAME(dataname)-
        KEYS(60 0)-
        CISZ(4096)-
        RECSZ(200 1020)-
        FSPC(20 20))-
        INDEX (NAME(indexname)-
        CISZ(2048))
```

Task 4—Allocate the VSAM Dataset to SOLVE

For MVS and MSP, add the dataset to the JCL, using a DD statement:

```
//dbname   DD DSN=clustername,DISP=OLD
```

Then stop the SOLVE region, job, or started task and restart it to pick up the extra DD statement.

Alternatively, the ALLOCATE command can be used to dynamically allocate the dataset to an active SOLVE system:

```
ALLOC DSN=clustername DD=dbname DISP=OLD
```

For VM/GCS, a DLBL statement can be added to the SOLVE procedure:

```
DLBL dbname V DSN clustername (VSAM)
```

Alternatively, the ALLOCATE command can be used to allocate the dataset to an active SOLVE system:

```
ALLOC DD=dbname DSN=clustername  MODE[CAT=catname]
```

For DOS/VSE, a DLBL statement can be added to the JCL:

```
// DLBL dbname,'clustername ,,VSAM,CAT=catname
```

Alternatively, the ALLOCATE command can be used to allocate the dataset to an active SOLVE system:

```
ALLOC DD=dbname DSN=clustername [CAT=catname ]
```

Task 5—Open the VSAM Dataset

Use the UDBCTL command to open and initialize the dataset. Whether the database will use the VSAM LSR pool depends on the options specified on the UDBCTL command.

By convention, the database name is the same as the DD name (MVS), filename (VM/GCS), or DLBL name (DOS/VSE).

If you do not want the database to use the LSR pool, issue the following UDBCTL command:

```
UDBCTL OPEN=dbname ID=* STRNO=7 BUFNI=10 BUFND=10
```

The values for STRNO, BUFNI, and BUFND are the suggested defaults.

The STRNO value should be 3 + (value of NDBSUBMX SYSPARM - 1, times 2). A lower value can lead to string space being dynamically acquired by VSAM (in MVS), or string waits (DOS/VSE or VM/GCS), if several &NDBSCAN statements are executing concurrently.

If you want the database to use the LSR pool, issue the following UDBCTL command:

```
UDBCTL OPEN=dbname ID=* LSR
```

Use the LSRPOOL command to define the LSR buffer pool sizes and the number of buffers for each size. The LSRPOOL command is documented in the *Management Services Command Reference.* LSR is the recommended way of running an NDB.

You might want to use deferred I/O when running the NDB. Deferred I/O involves sharing of buffers in between requests and enhances performance but possibly at the expense of integrity. Deferred I/O is *not* recommended when running an NDB in an online transaction update environment.

If you wish to run deferred I/O you must use the DEFER option of the UDBCTL command to open the file:

```
UDBCTL OPEN=dbname ID=* LSR DEFER
```

You must also use the DEFER option on the NDB START command. See the NDB START command description in the *Management Services Command Reference* for more information.

Task 6—Create the NDB

To do this, issue the NDB CREATE command:

```
NDB CREATE dbname [LOGSIZE=n] [LOADMODE] [LANG=lc]
```

The NDB CREATE command formats the UDB into an NDB by inserting control records that identify it as an NDB, and builds journal records for transaction management.

The nominated dataset (UDB, and so on) *cannot* be created into an NDB unless the dataset is empty and it meets the requirements such as KSDS, keylength, and record length. See Tasks 1 to 3 in this chapter and the NDB CREATE command description in the *Management Services Command Reference* for more information.

The NDB can be created as language-specific by specifying the LANG= operand. The uppercase translation table for the language specified is then used for all uppercase processing.

Specify the number of log blocks to format with the LOGSIZE parameter. If you wish to fast-load data, use the LOADMODE option to put the database into load mode.

Following the NDB CREATE command, the dataset is now formatted as an NDB, with no field definitions and no data records. From this point on, the other NDB command options and the &NDB*xxx* verbs can refer to it.

Task 7—START the Database

To allow NCL access to the database, the NDB START command must be used to keep the database active:

```
NDB START dbname [DEFER | NODEFER] [LOADMODE]
```

If the database is to be bulk-loaded (described in Task 9), consider using the DEFER option of the NDB START command. This option tells the database manager *not* to flush buffers after each update command (including &NDBADD, &NDBDEL, &NDBUPD), which improves performance at the expense of integrity. The DEFER option of the NDB START command is effective only if the database was opened with the UDBCTL command using the LSR and DEFER options. The LOADMODE option can be used to indicate that bulk-loading is to occur.

Task 8—Insert Field Definitions

An NDB cannot be used without field definitions. Field definitions can be added to or deleted from an NDB, using the NDB FIELD command at any time. Just after the successful completion of an NDB CREATE followed by an NDB START is a good time to add field definitions.

The following example shows how to add field definitions:

```
NDB FIELD  dbname ADD=SURNAMEFMT=C KEY=Y +
           NULLFIELD=N NULLVALUE=N
NDB FIELD  dbname ADD=FIRSTNAMEFMT=C KEY=N
NDB FIELD  dbname ADD=DOBFMT=D KEY=N
NDB FIELD  dbname ADD=ADDR1FMT=C KEY=N
NDB FIELD  dbname ADD=ADDR2FMT=C KEY=N
NDB FIELD  dbname ADD=ADDR3FMT=C KEY=N
NDB FIELD  dbname ADD=ADDR4FMT=C KEY=N
NDB FIELD  dbname ADD=SEXFMT=C KEY=N
NDB FIELD  dbname ADD=NAME
```

Note

The definitions could also have been added by using the &NDBDEF verb.

If the database is to have a *sequence key*, then the definition for it *must be added first*. A sequence key is defined by specifying KEY=SEQUENCE (can be abbreviated to KEY=S) on the field definition. A sequence key is forced to have the attributes UPDATE=NO, and NULLFIELD=NO. Records must always have a unique value for the sequence key field (like KEY=UNIQUE). A sequence key field definition cannot be deleted.

Task 9—Load Initial Data

If the database needs to have data loaded into it (for example, a table or reference database), then the data can be loaded using an NCL procedure to read the input data (for example, from a UDB, either a KSDS or an ESDS), and use &NDBADD to add it to the NDB.

If a large amount of data is to be loaded this way, then use the DEFER option of the NDB START command to prevent buffer flushing during the load.

When the load completes successfully, use an NDB START NODEFER command to remove the defer status. There is no need to stop and restart the database.

Another way to speed up loading of large amounts of data is to create or start the database in LOAD MODE. In this case, no keys are manipulated while loading, making the load run much faster.

The NDB ALTER command must then be used to build all keys in a single pass.

Note

If a system or SOLVE failure occurs while an NDB is open in DEFER mode, then it is flagged as unusable, and must be restored or recreated.

If the database is in LOAD MODE, it must have an NDB ALTER command run against it to build keys.

The load program should use the EXCLUSIVE option of the &NDBOPEN statement to prevent other users accessing the database while it is running.

The progress of the load program can be monitored by periodic use of the SHOW NDB=*dbname* command. This command displays information about the database, including the number of database requests executed (this is the sum of NDB commands and &NDB NCL statements) since it last started.

This completes the process of initialization (and initial loading) of an NDB. The tasks are the same as required for any other UDB, except for the NDB CREATE and field definition tasks.

Most of the initialization tasks can be combined into one NCL procedure, using &INTCMD/&INTREAD to issue and check the commands, and &NDB*xxx* statements to perform the definition and load. In an MVS (or MSP) environment, UTIL0007 can be used to perform the VSAM DELETE/DEFINE from NCL.

Deleting an NDB

An NDB can be deleted by using the standard VSAM (IDCAMS) DELETE command:

```
DELETE clustername CL
```

If the NDB is in use, then it must first be closed. To accomplish this, issue the following commands:

```
NDB STOP dbname IMM LOCK
UDBCTL CLOSE=dbname
UNALLOC DD=dbname
```

The NDB STOP command, with the IMM and LOCK operands, immediately stops the database, if it is active, and locks it from further access by any NDB commands or &NDB*xxx* statements. Any currently signed on users are given response 250 on their next request.

The UDBCTL CLOSE command physically closes the dataset.

The UNALLOC command frees the dataset so that it can be deleted.

Following the successful completion of the above commands, the IDCAMS DELETE can then be issued.

If the NDB is being deleted in preparation for reuse as an empty database, then the NDB RESET command provides a more convenient way to do this. Alternatively, if the dataset was defined to VSAM with the REUSE option, then a UDBCTL OPEN RESET command causes VSAM to clear the dataset back to empty.

Deleting All Data in an NDB

If an NDB is being used as a journal file (that is, it is cleared regularly after a specified time or number of records, for example), it can be cleared in three ways:

- Write an NCL procedure that reads the *entire* database sequentially (for example, by RID) and deletes all records—this method is slow and tedious.
- Physically delete the dataset (or issue a UDBCTL OPEN RESET command if the dataset is defined with REUSE), and re-create it, as described earlier in this chapter—this method has practical uses, such as when the dataset must be relocated on DASD or needs more space.
- Use the NDB RESET command—this method is normally the best approach.

  The NDB RESET command deletes all data records (and their keys) from an NDB, *but preserves the field definitions*. Thus, it is equivalent to deleting, defining, re-creating, and reissuing all &NDBDEF ADD statements or NDB FIELD commands required to build the field definitions.

  An NDB must not be active when the NDB RESET command is issued. This prevents active users from having the database cleared underneath them. A LOCKED database cannot be reset.

  The following sequence of commands resets an NDB:

  ```
  NDB STOP dbname IMM
  NDB RESET dbname
  ```

  If the database is not active, then the NDB STOP command effectively does nothing. The RESET command gives you the option of placing the database into LOAD MODE.

Altering Field Definitions in an NDB

Altering field definitions can be broken into three activities:

- Adding new field definitions to an NDB
- Deleting field definitions from an NDB
- Updating field definitions in an NDB

The first two can be done directly, at any time, even while users are accessing the database. Use the NDB FIELD ADD= *fieldname* and NDB FIELD DELETE= *fieldname* commands.

Adding New Field Definitions

Adding a field definition makes the field immediately available to all users.

Note

When adding field definitions *after* data is already in an NDB, defining a field with NULLFIELD=NO causes errors on the update of any record that existed prior to the definition, if the newly defined field is not included in the update list. This is because it is a required field, and is not in the record.

Deleting Field Definitions

Deleting a field definition logically removes the field from the database. Following the delete, all data values for that field become inaccessible.

Note

Deleting a field immediately makes the field inaccessible to NCL procedures. Any currently active sequences (&NDBSEQ) defined on that field, if it is keyed, are given a response code on the next &NDBGET referring to that sequence. Predefined formats (&NDBFMT) referring to the field still valid until they are redefined. An in-progress &NDBSCAN could get undefined results.

Deleting a field causes a physical VSAM delete to be performed for all the associated key records. The data records, however, are not updated. To do so would create an unacceptable overhead. Instead, the field value in each data record is regarded as *logically* deleted. It is inaccessible. Whenever a data record is updated, all logically deleted fields are removed.

Updating a Field Definition

A field definition can be updated at any time. However, not all field attributes can be changed at any time. Some changes require the database to be empty (that is, just created or reset), or in LOAD MODE. Some field attribute alterations are prohibited.

The following field attributes can be changed at any time:

- DESC = *description*
- USER1, USER2, USER3, USER4
- NULLVALUE
- NULLFIELD (except for KEY = SEQ field)
- UPDATE (except for KEY = SEQ field)
- KEY = Y to KEY = N
- KEY = U to KEY = Y or N
- NEWNAME (to rename the field)

The following attribute can be changed if the field is not keyed (that is, KEY=NO):

- CAPS = YES to CAPS = SEARCH

The following attributes can be changed if the database is in the LOAD MODE or empty:

- KEY (except to/from SEQ)
- CAPS = SEARCH/NO to CAPS = NO/SEARCH

The following attributes can be changed if the database is empty:

- FMT
- CAPS
- KEY (except to/from SEQ)

The following attribute *cannot* be changed:

- KEY = SEQ (to or from)

Note

A field can also be changed from KEY = N to KEY = Y by using the NDB ALTER command.

Backing Up an NDB

Back up an NDB with the standard IDCAMS utility functions (that is, REPRO or EXPORT). For integrity, the dataset should not be open. Issue the following commands to ensure this:

```
NDB STOP dbname IMM LOCK
UDBCTL CLOSE=dbname
```

Following completion of the backup, issue the following command to make the NDB available again:

```
UDBCTL OPEN dbname options
NDB START dbname UNLOCK options
```

The options on these commands should agree with any standard options specified (for example, LSR/DEFER on the UDBCTL statement).

Restoring an NDB

If an NDB must be restored from a backup copy, then use the standard IDCAMS restore facility associated with the backup copy. For example, if REPRO was used to backup the database, then use REPRO to restore it; if EXPORT was used to backup the database, then use IMPORT to restore it.

When using REPRO, define the database with REUSE to allow the REUSE option of REPRO to be used. This makes a DELETE/DEFINE prior to the restore unnecessary.

The database must not be open while a restore is in progress. To create a consistent environment for NCL procedures using the NDB, issue an NDB STOP LOCK command prior to the restore. This causes all &NDBOPEN statements executed to get a *database locked* response.

Upon completion of the restore, issue the UDBCTL OPEN, as required, and an NDB START UNLOCK command to restart the database, and release the lock.

Monitoring NDB Activity

Use the SHOW NDB, SHOW NDBUSER and the TRACE option of the NDB START command to monitor NDB activity.

To determine the number of NDBs currently active, issue the command:

```
SHOW NDB
```

The output from this command shows the number of NDBs active, locked, and stopping. To obtain detailed statistics about all active or locked NDBs, or a specific NDB, use the command(s):

```
SHOW   NDB=ALL        -* for all, or
SHOW   NDB=dbname     -* for a specific NDB.
```

The output from these commands indicates the status of the NDB(s), the number of signed on users, the number of commands processed, and whether the database was started in DEFER status.

This display is particularly useful for monitoring the progress of long-running procedures that are loading or reading large numbers of records.

To determine the user ID(s) of users signed on to all active NDBs, issue the command:

```
SHOW NDBUSER
```

The output from this command, which is similar to the SHOW UDBUSER command, indicates details such as the database, user ID, and LU name (terminal) for all users signed on to any NDB.

Note

A SHOW UDBUSER shows *NDB as the only user of a UDB that is an NDB. This serves as an indication to the issuer of the SHOW UDBUSER command that a SHOW NDBUSER command is required for this UDB/NDB to obtain a list of signed on users.

Using the TRACE option of the NDB START command causes a message to be written to the SOLVE activity log every time an &NDB verb or an NDB command is issued involving that particular NDB. Tracing can be stopped at any time by issuing an NDB START command specifying NOTRACE.

Monitoring NDB Performance

Monitoring is used, for example, for physical tuning and buffer tuning. Use a periodic IDCAMS LISTCAT command to monitor the physical attributes of the NDB, including DASD space, number of extents, and CI splits.

An NDB itself does not need reorganization internally. VSAM, however, might need to reorganize to remove excessive CI and/or CA splits if a large amount of key addition, deletion, or value changing takes place. This can be accomplished by using standard IDCAMS services to backup, delete, redefine, and restore the dataset.

Warning

You must not change the VSAM key length or maximum record length during this reorganization. If you do, the NDB will not be usable.

If you need to increase the key length, the NDB must be *logically* unloaded, deleted, re-created, and *logically* reloaded. Chapter 19, *Using &NDBSCAN Statements*, includes an example of how to accomplish a logical unload and reload.

The SHOW VSAM command allows you to determine whether the NDB performance would benefit from increased buffering, if not running from the LSR pool, or from increases or changes in the LSR pool definition. Standard VSAM tuning techniques should be followed.

Improving Performance by Using LOAD MODE

As NDBs use inverted-list indexes, bulk record addition, update, or deletion can be slow. This is because of the large number of physical file updates required to manage all the keyed field indexes.

To allow fast loading (in particular), an NDB can be placed in LOAD MODE. In LOAD MODE, no keys (other than the optional sequence key) are maintained. This greatly improves record add speed.

When in LOAD MODE, records can be added, updated, deleted, retrieved, or scanned. However, none of these operations use keys. This means that GET or SEQ by a key is not allowed. Only RID access is permitted.

The only way to take an NDB out of LOAD MODE is to use the NDB ALTER command. This command, using the BLDX DB option, reads all the NDB data records, extracts all keys, sorts them, and writes them in a single pass. It checks for errors (for example, unique key violation) and, when finished, resets LOAD MODE.

Checking an NDB for Consistency

If you want to validate the relationship between keys and data in an NDB, you can use the NDB ALTER command CHKX option to do this. It extracts and sorts all keys from the data and then compares them with the actual key records in the NDB. All errors are reported.

Multiple System Access to an NDB

An NDB may be corrupted by:

- The database being opening by two regions (same machine or shared DASD)
- The database being opened twice (under different file IDs) on the same region

There are several internal protection mechanisms that attempt to prevent such corruption:

- The VSAM timestamp of the NDB is compared with all other open NDBs. A match prevents the NDB from opening. This blocks one NDB from being opened twice by the same region.

  Note

  It is not totally foolproof, as VSAM updates the timestamp each time the VSAM dataset is physically closed (that is, UDBCTL CLOSE occurs).

- While an NDB is open, the control record contains the domain ID of the accessing Management Services. This field is closed when the NDB is closed.

  If, when opening an NDB, a non-blank domain ID is found, and it is not the same as the current domain ID, then the NDB does not open unless the FORCE operand of the NDB START command is used.

  This prevents a database being opened by two different SOLVE regions. Use the FORCE operand if there is a system failure and the NDB must be opened on a different region.

- The name used to open the NDB (that is, UDBCTL ID = *operand value*) is stored in the NDB control record. If a mismatch occurs on open, a warning message is written. The open is allowed to continue unless other problems occur.

- In spite of the previous precautions, an NDB can always be opened in INPUT MODE. This bypasses all the above checks but no updates can be performed.

- NDB security—NDBs can be protected by the NCL security exit, NCLEX01. This protection can be used to restrict users' NCL procedures to specific NCL statements for functions such as updating and adding. Full details are provided in Appendix H, *Using NCLEX01 for NDB Security*.

Using the NDB Journal

Follow these tasks to implement NDB journaling on your region.

Task 1—Define Two Journal Datasets

Use IDCAMS to define two VSAM Entry Sequenced Datasets (ESDS). The following parameters are required (use sample $NDIDCDJ):

NONINDEXED	Indicates an ESDS.
REUSE	Journal is cleared when opened for update.
RECORDSIZE	Journal record size should be greater than the recordsize minus keylength + 60 of the largest NDB using the journal (see below).

The space allocated to the journal depends on the number of NDB updates occurring on the system, and the frequency with which you swap journals for batch processing. The volume should ideally be on a separate unit to the NDBs being journaled.

Task 2—Allocate Datasets to Management Services

Allocate the two journal datasets, using DISP=SHR, to Management Services by either including them in the JCL, declaring them in the UTIL0028 SYSIN deck, or allocating them in NMINIT or NMREADY NCL procedures.

Task 3—Make Duplicates of the NDBs

Stop the NDBs you want to journal and create duplicates of them using IDCAMS REPRO.

Note

The duplicate must be created after the NDB CREATE has been performed to initialize the NDB.

Once the duplicate has been created, it is important to keep it synchronized with the primary copy by ensuring that all NDB journal records written after the duplicate was made are applied to it.

For an existing NDB which has already been loaded, the decision to start journaling updates can be made at any time.

If you regularly shut down your Management Services system, it is recommended that you resynchronize your NDB backups with the primary copy. Do not apply journals created before the new backup copy.

Task 4—Add the Duplicates to the Batch Forward Recovery JCL

The batch forward recovery utility (UTIL0010) applies the NDB journal to a duplicate copy of the NDBs.

Add the dataset name of the NDB duplicate to your batch forward recovery JCL (use sample $NDUT010).

Make sure you have updated the JOURNL1 and JOURNL2 DD statements in the sample with the two journal dataset names you defined in Task 1.

Include in the utility JCL all NDBs which require forward recovery. By default, the utility attempts to forward recover all NDBs which have journal records present in the journal dataset. The DD name for the NDB should be the same as the NDB name used to identify it to Management Services.

Use the control cards on the U10IN dataset to specify a subset of NDBs.

The format of a U10IN control card is:

```
RECOVER NDB=ndbid [,DD=name]
```

where the DD operand can be used to specify an override DD name for the NDB.

`RECOVER ALL` is used to indicate that all NDBs are to be forward recovered. This is the default.

The following is a sample JCL specification for the forward recovery utility:

```
//FWDRECVR EXEC PGM=UTIL0010,PARM='JOURNAL=JOURNL1'
//STEPLIB DD DSN=steplib,DISP=SHR
//U10PRINT DD SYSOUT=*
//U10IN DD*
  RECOVER ALL
//JOURNL1 DD DSN=SOLVE.NDB.JOURNAL1,DISP=SHR
//JOURNL2 DD DSN=SOLVE.NDB.JOURNAL2,DISP=SHR
//*
//NDB1 DSN=SOLVE.NDB.BACKUP.NDB1,DISP=SHR
//NDB2 DD DSN=SOLVE.NDB.BACKUP.NDB2,DISP=SHR
//NDB3 DD DSN=SOLVE.NDB.BACKUP.NDB3,DISP=SHR
//*
```

Note

The JOURNAL=JOURNL1 parameter specified on the EXEC UTIL0010 statement is used to override the journal DD name. If this parameter is not specified, then the DD name used is JOURNAL.

Task 5—Start the NDBs

Issue an NDB START command with the JOURNAL operand specified.

The first time an NDB START command (with the JOURNAL operand) is issued, it causes the journal datasets to be opened. The system normally checks both journal datasets and begins processing with the oldest one. This gives you time to run the forward recovery utility if the system was restarted after failure.

If JOURNL1 is empty, then no attempt is made to check the second dataset, and journaling begins immediately on JOURNL1.

NDB Journal Swapping

Journals swap if a JOURNAL SWAP command is issued or if the journal in use runs out of space.

Each time a journal is swapped, the journal control NCL procedure is started to assist the automation of forward recovery. The SYSPARMS JRNLPROC command can be used to specify the procedure name (the default is $NDJPROC).

If the Management Services system terminates abnormally, then you must submit the NDB forward recovery job manually (or from an alternate automation process), as the other journal is used after the restart, and a swap effectively occurs.

Forward recovery must be performed before a journal swap occurs after the system restart.

It is recommended that you apply journals to the NDB backups as soon as they are swapped.

18

Using &NDB Verbs

This chapter provides examples of the use of the NCL verbs related to NetMaster Databases (NDBs). There are examples that include adding, deleting, updating and retrieving records, as well as examples that use information about the database itself.

The success or failure of many &NDB verbs is indicated as a completion or error code returned in the system variables &NDBRC and &NDBERRI on completion of the function. Appendix F, *NDB Response Codes*, contains a full list of the possible values of &NDBRC and &NDBERRI. Throughout this chapter, reference is made to specific return codes that can occur on particular conditions.

The full syntax of the &NDB verbs is described in the *Network Control Language Reference* manual.

This chapter discusses the following topics:

- Relationship Between &FILE and &NDBxxx Verbs
- Accessing an NDB
- Working with NDBs
- Notes on Sequential Retrieval
- Obtaining Information About an NDB
- Changing NDB NCL Processing Options
- Putting It All Together—Unloading/Reloading an NDB

Relationship Between &FILE and &NDB*xxx* Verbs

Although the &NDB verbs are functionally similar to the &FILE verbs, there are some significant differences. See Table 16-1 on page 16-4 for a list of these differences.

Protecting Your Data Values with &NDBQUOTE

Several &NDB verbs use a free-form text syntax. This syntax is described fully in the &NDBSCAN verb description in the *Network Control Language Reference.*

When processing the free-form text, certain characters, for example, '(', ',' and ')', act as delimiters. If the data value contains one of these characters, it could cause a syntax error, and will not be preserved.

To prevent this, data values can be *quoted*, using either single (') or double (") quotes. If the data contains the selected quote character, two of that character must be used.

The &NDBQUOTE built-in function *automatically* quotes the data *when required*. It knows the characters that require quoting, and picks single or double quotes as required. It also handles doubling of embedded quotes.

The examples in this chapter illustrate use of the &NDBQUOTE built-in function where appropriate.

Preserving Lower Case Data

If data to be stored in an NDB is to contain lower case characters, ensure that &CONTROL NOUCASE is in effect in all procedures that use the &NDBADD or &NDBUPD verbs. Failure to do so could mean the accidental folding of data to upper case. This also applies to &NDBDEF and the NDB FIELD command when adding field descriptions or USER1-4.

Defining and Deleting Fields in an NDB

An NDB has no predefined fields. All fields must be defined *once* before they can be used. This is normally done by the Database Administrator when the NDB is defined. See the section, *Altering Field Definitions in an NDB* in the chapter, Chapter 17, *Netmaster Database (NDB) Administration*.

Field definitions can be added or deleted *at any time*, even while other users are accessing the NDB, although you might not want to do this.

The names and attributes of defined fields can be obtained using the &NDBINFO verb (see the section titled *Obtaining Information About an NDB*, on page 18-14).

Accessing an NDB

Before any records can be added to, deleted from, or retrieved from an NDB, the NCL process must sign on, or *connect* itself to the NDB. This is accomplished using the &NDBOPEN verb:

```
&NDBOPEN dbname
```

This statement registers the NCL process as a *signed-on user* of the nominated NDB.

If you want the NCL process to have exclusive control over the NDB, then specify the keyword, EXCLUSIVE, after the database name.

If you want to perform input only operations, then use the INPUT keyword .

If a security exit is being used, the DATA keyword can be used to allow up to 50 characters of data to be passed to the exit.

The &NDBRC system variable should always be checked after &NDBOPEN, to ensure it was successful. If the value of &NDBRC is not 0, then the open has failed, except for response 34, which indicates that the NCL process is already signed on.

When an &NDBOPEN is executed, &NDBCTL ERROR=CONTINUE is always assumed, to allow the NCL process to handle errors at this point. If an open failure occurs, and no action is taken, then the next &NDB*xxx* statement for that database will get a *not open* response. Response 34 (already open) *will* terminate the procedure unless an explicit &NDBCTL ERROR=CONTINUE is in effect.

The &NDBOPEN verb is analogous to the &FILE OPEN verb. However, there is no need to reissue &NDBOPEN after referencing an NDB of another name. This is because &NDB statements that refer to an NDB require the name of the NDB that is to be accessed, whereas the &FILE verbs assume the *UDB named in the last executed* &FILE OPEN statement.

EASINET Considerations

If NDBs are being accessed by EASINET procedures, they *should not* be open across the &PANEL statement that displays the network logo. To do so would increase the amount of storage *each* network terminal would need.

Closing an NDB

When an NCL process has completed use of an NDB, use the &NDBCLOSE verb to sign off (*disconnect*) the NCL process from the NDB:

```
&NDBCLOSE dbname
```

The &NDBCLOSE verb disconnects the NCL process from the nominated NDB, if it is connected (signed on).

All storage associated with the process is freed. This includes storage for defined formats (&NDBDMT), sequences (&NDBSEQ), and scan result lists (&NDBSCAN).

&NDBCLOSE is analogous to the &FILE CLOSE verb.

Working with NDBs

You can use NCL verbs to add records to and update, delete, and retrieve them from an NDB.

Adding Records to an NDB

The &NDBADD verb is used to add records to an NDB. The fields for the new records must be named on the &NDBADD statement, along with their values. *Not all fields need to be supplied*. For the database manager, only those fields with the attribute NULLFIELD = NO need be supplied—this includes the sequence key, if one is defined.

If you only need to supply a few fields on the &NDBADD statement, code it as follows:

```
&NDBADD dbname DATA name1 = value1 name2 = value2 ...
```

If you need to supply a large number of fields, or you are not sure of the exact content of the new record (for example, table driven systems), the second format of the &NDBADD verb can be used:

```
&NDBADD dbname START
&NDBADD dbname DATA name1 = value1
&NDBADD dbname DATA name2 = value2
&NDBADD dbname DATA name3 = value3
&NDBADD dbname DATA name4 = value4
&NDBADD dbname END
```

This syntax allows a record of *any length* to be created.

The fields need not be in any order, and the collection of &NDBADD statements *need not* be adjacent. For example, a loop could be used, with complex substitution, to build the list of field names and values:

```
&NDBADD MYNDB START
&I = 1
&DOWHILE &I LE &NFLDS
   &VALUE = &NDBQUOTE &FV&I
   &NDBADD MYNDB DATA &FN&I = &VALUE
   &I = &I + 1
&DOEND
&NDBADD MYNDB END -* this statement calls the DBMS
```

In this example, you loop through a table of field names and values, adding each field.

Following the &NDBADD, or &NDBADD END, if you are using START/DATA/END, the &NDBRC system variable contains 0 if no errors were encountered. If not 0, an error response indicates the problem, and &NDBERRI might have additional information. An error message is also displayed unless &NDBCTL MSG=NO is in effect.

If the response is 0, &NDBRID contains the RID assigned to the new record. This RID can be used in other &NDB statements to refer to the new record.

Updating Records in an NDB

A record in an NDB can be updated using the &NDBUPD verb. Unlike the &FILE PUT verb, an &NDBUPD verb only updates the nominated fields. All other fields retain their existing value. The &FILE PUT verb replaces the entire record.

The RID(s) of the record(s) to be updated must be known. Normally, the RID is determined by a preceding &NDBGET or &NDBSCAN.

To update just a few (fixed amount) fields, code:

```
&NDBUPD dbname RID=rid DATA name1 = value1 name2 = value2
```

where *name1*, *name2*, ... are the names of the fields to be updated, and *value1*, *value2*, ... are the values.

If you need to update a large number of fields, or you are not sure of the exact content of the update, an alternative format of the &NDBUPD verb can be used:

```
&NDBUPD dbname RID=n START
&NDBUPD dbname DATA name1 = value1
&NDBUPD dbname DATA name2 = value2
&NDBUPD dbname DATA name3 = value3
&NDBUPD dbname DATA name4 = value4
&NDBUPD dbname END
```

This syntax allows *any number* of fields to be updated.

The fields need not be in any order, and the collection of &NDBUPD statements *need not* be adjacent. For example, a loop could be used, with complex substitution, to build the list of field names and values:

```
&NDBUPD MYNDB RID=&UPDRID START
&I = 1
&DOWHILE &I LE &NFLDS
   &NDBUPD MYNDB DATA &FN&I = &VALUE
   &I = &I + 1
&DOEND
&NDBUPD MYNDB END -* this statement calls the DBMS
```

In this example, we loop through a table of field names and values, updating each nominated field.

Note

Fields defined with UPDATE=NO can be specified in the update list, as long as the same value as is currently in the record is supplied.

A field that is to be set to null (not present) can be represented by coding:

```
fieldname = &NULL
```

where &NULL is an undefined variable. If the field definition has NULLFIELD=NO, an error response will be given.

Following the &NDBUPD or &NDBUPD END, if you are using START/DATA/END, the &NDBRC system variable contains 0 if no errors were encountered. If not 0, an error response indicates the problem, and &NDBERRI might have additional information. An error message is also displayed unless &NDBCTL MSG=NO is in effect.

If the response is 0, &NDBRID contains the RID that was supplied on the &NDBUPD verb.

Deleting Records from an NDB

To delete records from an NDB, use the &NDBDEL verb. The RID(s) of the record(s) to be deleted must be known. The RID is normally obtained from a preceding &NDBGET or &NDBSCAN.

To delete a record, code:

```
&NDBDEL dbname RID=&rid
```

where *&rid* is an NCL variable that contains the RID.

There is no direct equivalent to the &FILE DEL KEQALL/KGEALL generic delete. However, embedding an &NDBDEL in a loop controlled by an &NDBGET GENERIC or &NDBGET SEQUENCE=*name* allows easy deletion of any group of records (particularly powerful after an &NDBSCAN). For example, to delete all records that have SURNAME = SMITH and STATUS = DEPARTED, code:

```
&NDBSCAN MYNDB SEQ=S1 DATA SURNAME = SMITH AND STATUS = +
                      DEPARTED
&IF   &NDBRC = 0 &DO
   &NDBGET MYNDB SEQ=S1 FORMAT NO-FIELDS
   &DOWHILE &NDBRC = 0
      &NDBDEL dbname RID=&NDBRID
      &NDBGET dbname MYNDB SEQ=S1 FORMAT NO-FIELDS
      &DOEND
&DOEND
```

Following the &NDBDEL, the &NDBRC system variable is set to 0 if the delete was successful, or 1 if no record with the supplied RID was found.

Retrieving Records from an NDB

To retrieve records from an NDB, use the &NDBGET verb. There are two related verbs:

- &NDBFMT, used to predefine a list of the fields to be returned by &NDBGET
- &NDBSEQ, used to define a sequential read path for an NDB

You can retrieve records from an NDB in several ways:

- Direct by RID, allowing EQ, GE, GT, LE, LT relationships with your RID value. This is useful when you know the RID of the record you want.
- Direct by *any* keyed field. The key field value of the returned record can be EQ, LE, LT, GE, GT, or generically equal to the supplied key. If the key field is not unique, only the record with the lowest RID for any set of records with the same key field value can be accessed this way.
- Sequentially by RID. Records are returned in ascending or descending RID sequence. For databases without a sequence key, this is the fastest way to read the entire database. Optionally, a starting and/or ending RID can be nominated. The last record read can also be re-read without knowing its RID.
- Sequentially by *any* key field (including a sequence key, if defined). Records are returned in ascending or descending key field sequence. If the key is not unique, RID is used to sequence records with an equal key field value. For databases defined with a sequence key, retrieval by that field is the fastest way to read the entire database. Optionally, a starting and/or ending key value can be nominated. The last record read can also be re-read without knowing its RID.
- Sequentially from a list of records that passes an &NDBSCAN. This option is covered fully in Chapter 19, *Using &NDBSCAN Statements*.

- Indirectly by a keyed field where its value is stored in another record in the NDB.
- You can also retrieve statistical information about any key field (also known as a *histogram*).

Each of the above techniques is described in detail below.

Defining Fields to Return (&NDBFMT)

When retrieving records from an NDB, it is likely that not all the fields in a record are needed. Also, it might be desirable to return the fields in NCL variables with different names. The &NDBFMT verb allows you to define any number of *named* formats, that can be nominated on an &NDBGET statement, referring to the same NDB. The nominated format determines which fields will be returned, and the names of the NCL variables that will receive the values.

The defined format can request that all fields defined in the database be returned (ALL-FIELDS), or that all fields defined as keyed be returned (KEY-FIELDS), or that no fields be returned (NO-FIELDS) (useful for just checking for existence of a record), or that a list of nominated fields be returned, optionally in NCL variables with a different name.

Note

A format can also be specified directly on the &NDBGET statement, but this is not recommended, particularly if the &NDBGET statement is in a loop. The format definition must be parsed and encoded every time the &NDBGET statement is executed.

A format stays defined until explicitly deleted (by &NDBFMT ... DELETE), or until an &NDBCLOSE is executed for that NDB.

Examples of predefined formats are shown below:

```
&NDBFMT MYNDB DEFINE FORMAT=FMT1 DATA ALL-FIELDS
&NDBFMT MYNDB DEFINE FORMAT=FMT2 DATA NO-FIELDS
&NDBFMT MYNDB DEFINE FORMAT=FMT3 START
&NDBFMT MYNDB DATA FIELDS
&NDBFMT MYNDB DATA SURNAME FIRSTNAME
&NDBFMT MYNDB DATA ( LNAME = LASTNAME)
&NDBFMT MYNDB END
```

It is evident that the START/DATA/END syntax, as described under &NDBADD and &NDBUPD, is also available when using &NDBFMT. Thus, complex variables and open-ended formats can be defined. The START/DATA/END syntax *is not* available on the &NDBGET verb. Any format defined in an &NDBGET statement must fit on the &NDBGET statement itself.

The examples below all assume predefined formats.

Determining Which Fields Are Present

Since NDBs support null fields, it might be necessary to determine, when a particular record is read, which fields are present and which are not. The MODFLD option on the &NDBGET verb allows the use of &ZVARCNT and &ZMODFLD to access the modified fields:

```
&NDBGET dbname retrieval-method format-method MODFLD=YES
&field1 = &ZMODFLD
&field2 = &ZMODFLD
...
```

Retrieving Records Directly by RID

When you know the RID of the record that you want, perhaps from an earlier &NDBADD, you can retrieve it by coding:

```
&NDBGET dbname RID=ridvalue FORMAT=fmtname
```

ridvalue is a variable containing an RID obtained elsewhere.

You can also code OPT=KGE, and so on, to retrieve a record with an RID satisfying the coded option. For example, OPT=KGT returns the record with the next-highest RID than the value passed. It is possible to simulate sequential reading by RID using this technique.

Retrieving Records by Key Field

When you know the value for a key field, retrieve a record matching that key field by coding:

```
&NDBGET dbname FIELD=fieldname VALUE=value FORMAT=fmtname
```

Code OPT=KGE to return a record where the key field satisfies the coded option. For character or HEX fields, OPT=GENERIC is also supported. Use OPT=KGT for forward, or OPT=KLT for backward, and supply the last key field value as the search argument to simulate sequential reading.

For key fields that are not unique, this technique always returns the record with the lowest RID, from the set of records having the same key field value. The other records *cannot* be accessed this way.

Reading Sequentially by RID

When the entire database, *or* a large portion of it, must be read sequentially, and the order of returned records is unimportant, the following technique should be used.

```
&NDBSEQ dbname DEFINE SEQ=SEQ1 RID
&NDBGET dbname SEQ=SEQ1 FORMAT=fmtname
&DOWHILE &NDBRC = 0
   ...  process record
   &NDBGET dbname SEQ=SEQ1 FORMAT=fmtname
&DOEND
```

The &NDBSEQ statement sets up a sequential retrieval definition, that can be used on an &NDBGET statement to retrieve records sequentially by RID. By default, the options SKIP=+1 and DIR=FWD are assumed on the &NDBGET, so the database is read from lowest RID to highest RID.

Options on the &NDBSEQ statement allow specification of a low and high RID.

If the database has a sequence key, the next method will perform better.

Retrieving Records Sequentially by Key Field

When you wish to retrieve records by *any key sequence*, the following technique can be used:

```
&NDBSEQ dbname DEFINE SEQ=seqname FIELD=keyfieldname +
          FROM=fromvalue TO=tovalue
&NDBGET loop as coded in previous example.....
```

This use of &NDBSEQ and &NDBGET allows sequential retrieval by *any* key field, and, for non-unique key fields, records with the same key field value are returned in RID sequence within the value.

The FROM and TO operands on &NDBSEQ are optional. If omitted, the lowest (FROM) or highest (TO) key values are used. For non-unique key fields, VALUE=*value* can be coded to indicate the from and to values are the same. This is useful when you want to read all records with the same non-unique value. For character and HEX fields, GENERIC=*value* can be coded to indicate the range is to cover all records with that generic key value.

Retrieving Records Indirectly by Key Field Stored in Another Record

If the NDB has multiple record types (see the section on null fields in Chapter 16, *NDB Concepts*), the key identifying the desired record is not always known immediately. The identifying key might be stored in the field of another record.

For example, if the NDB contains an ordering system, it might be necessary to access a customer record, given only an order number, the order record of which contains a key identifying a separate customer record.

NDBs allow this type of access through the .LINK option of the &NDBGET/&NDBFMT verbs. The technique is:

```
&NDBGET dbname RID=record ID FORMAT FIELDS .LINK +
(FROM=fldname TO=keyed-fieldname ) FIELDS formatlist
```

Up to 16 different records can be accessed in the one &NDBGET statement using linking; that is, one primary and up to 15 secondary records.

Any access method can be used for retrieving the first record. The retrieval method for subsequent records is equivalent to an OPT=KEQ get.

Retrieving Keyed Field Statistics (Histogram)

All keyed fields in an NDB (except the sequence key) contain statistical information on the number of records that have a particular unique key value.

The KEY option of &NDBGET and &NDBSEQ allow retrieval, not of a record, but of a *key value* and *record count*. Thus, statistical information can be obtained. This is very fast, as no data is accessed. When reading the NDB in this way, the format specified on &NDBGET is completely ignored. (You can, for example, code FORMAT = NO.) Instead, the data is returned in two NCL variables:

```
&NDBKEYVALUE keyed field value
&NDBKEYCOUNT keyed field record count
```

For example, to find out how many records have field *surname* with a value of ‘Smith’:

```
&NDBGET dbname KEY=SURNAME VALUE=‘SMITH’ FORMAT=NO
```

To get a sequential breakdown of a key field’s contents and counts:

```
&NDBSEQ dbname DEFINE SEQ=S1 KEY=SURNAME
&NDBGET dbname SEQ=S1 FORMAT=NO
&DOWHILE &NDBRC=0
          &WRITE &NDBKEYVALUE &NDBKEYCOUNT
          &NDBGET dbname SEQ=S1 FORMAT=NO
&DOEND
```

Notes on Sequential Retrieval

The &NDBSEQ verb (and &NDBSCAN verb, discussed in Chapter 19, *Using &NDBSCAN Statements*), allows you to specify a name for a sequence. Unlike the &FILE verbs, an NCL process can have *any number* of concurrent sequences defined on *any number* of databases. Positioning is handled internally, with no need to use accompanying number of VSAM strings.

The only overhead is an internal NDB control block that maintains positioning.

Performing a direct get *does not* destroy any positioning maintained by active sequences.

KEEP=YES on &NDBSEQ

By default, a sequence is automatically deleted when an end-of-file response is returned (&NDBRC = 2). This is normally what is wanted, and saves you coding an &NDBSEQ DELETE. If, however, you want the sequence to stay defined, you can code KEEP=YES on the &NDBSEQ DEFINE, and the sequence stays defined until explicitly deleted, or the NDB is closed.

&NDBSEQ RESET

A defined sequence can also be *reset*, allowing you to restart the sequence at any point. For example, after reading forward, and getting an EOF response, you are positioned after the last record in the sequence. Forward gets will continue to return the EOF response. To re-read from the beginning, issue an &NDBSEQ RESET as shown:

```
&NDBSEQ dbname RESET SEQ=seqname
```

Optionally, you can nominate a key value to reposition to. The next get will return the first record with the matching, or higher, or lower, if no match, and depending on the direction:

```
&NDBSEQ dbname RESET SEQ=seqname REPOS=value
```

&NDBGET DIR= and SKIP=

When reading sequentially, the &NDBGET verb allows you to specify the direction of retrieval, FWD or BWD, and a skip amount.

The default is DIR=FWD and SKIP=+1, which causes the next-higher key record to be read.

SKIP=0 causes a *re-read* of the last record read in that sequence. This is useful where you are retrieving only a subset of fields, and sometimes need to obtain extra fields. You need not specify a direct get by RID.

A skip amount (*n*) greater than 1 will allow you to skip *n-1* records on each &NDBGET. This is extremely useful when extracting samples from a database, and especially useful when scrolling in selection lists (see following examples).

A negative skip amount causes the specified or defaulted direction to be inverted. For example, SKIP=-5 is equivalent to SKIP=5 with DIR=BWD.

Specifying DIR=BWD allows a backward read to be performed. Thus, when you need to retrieve records in *descending* key sequence, DIR=BWD can be used.

Reading by Sparse Keys

When defining a sequence on a key field that is defined (or defaulted) with NULLFIELD=YES, it is possible that some records *do not contain the field*. These records will not be returned when reading by that sequence. This is because no index record is built for fields not present in a data record. This can be especially useful when the database contains multiple record types, as illustrated in one of the examples below.

Examples

The following example, shows how two sequences defined on a single database, where the logical record types are *disjoint* (as described in Chapter 16, *NDB Concepts*), using different key fields, can perform a master/transaction update:

```
&NDBSEQ MYNDB DEFINE SEQ=MAST FIELD=MASTKEY
&NDBSEQ MYNDB DEFINE SEQ=TRAN FIELD=TRANKEY
&GOSUB .READMAST  -* read mast, set &MASTKEY to 999999 if
                  -* eof
&GOSUB .READTRAN  -* read tran, set &TRANKEY to 999999 if
                  -* eof
&DOWHILE &MASTKEY.&TRANKEY NE 999999.999999
   &IF &MASTKEY = &TRANKEY &DO
            ...process match
      &GOSUB .READTRAN
   &DOEND
   &ELSE &IF &MASTKEY GT &TRANKEY &DO
            ...process unmatched transaction
      &GOSUB .READTRAN
   &DOEND
   &ELSE &DO
            ...process unmatched master
      &GOSUB .READMAST
   &DOEND
&DOEND
```

The next example shows how SKIP= can be used to extract a subset of a database, perhaps for statistical analysis.

```
&NDBSEQ  MYNDB DEFINE SEQ=S1 RID
&NDBINFO MYNDB DB    -* obtain # records in DB
&SKIP = &NDBDBNRECS / 1000 -* determine skip to get 1000
                     -* recs
&NDBGET  MYNDB SEQ=S1 SKIP=&SKIP FORMAT ALL-FIELDS
&DOWHILE &NDBRC = 0
               ...write sampled record.
   &NDBGET MYNDB SEQ=S1 SKIP=&SKIP FORMAT ALL-FIELDS
&DOEND
```

The next example shows how a selection list can be easily processed using a sequence and SKIP/DIR:

```
&NDBSEQ MYNDB DEFINE FIELD=SURNAME KEEP=YES
&SKIPVAL = 1      -* initial skip
.LOOP &GOSUB .BUILD_PANEL  -* builds panel.
                  -* position now is last record on screen
&PANEL XYZPANEL
&IF &INKEY = PF08 &THEN &DO
   &SKIPVAL = +1  -* skip to next record after bottom
   &GOTO .LOOP
&DOEND
&ELSE       &IF &INKEY = PF07 &DO
&SKIPVAL = (0-(&LUROWS*2))
   -* skip top + back 1
-*screens length
   &GOTO .LOOP
&DOEND
```

Obtaining Information About an NDB

The &NDBINFO verb allows you to obtain information about the current state of an NDB, or any of the defined fields in an NDB.

To obtain information about the *database*, code:

```
&NDBINFO dbname DB
```

This statement sets several user NCL variables, prefixed by &NDBDB, with database information. Some useful fields are described below. See the &NDBINFO verb description in the *Network Control Language Reference* for the full list of fields. Useful fields are:

&NDBDBNRECS

Contains the current number of records in the database

&NDBDBNFLDS
Contains the current number of fields defined in the database

&NDBDBVKL
Contains the VSAM keylength of the database

&NDBDBVRL
Contains the VSAM maximum record length of the database.

These fields can be used to determine if two NDBs are compatible (for example, for backup purposes).

The &NDBINFO verb also allows retrieval of information about fields defined in the database. Information can be retrieved about a field with a specific *name*, or by relative field *number*, where the number is from 1 to the value returned in &NDBDBNFLDS. This relative number is *not* fixed, but changes as field definitions are added and deleted. For this reason, inquiry by number should only be used in a loop (from 1 to &NDBDBNFLDS), with the database open EXCLUSIVE.

To retrieve information about a specific field, where the name is known, code:

```
&NDBINBFO dbname NAME=fieldname
```

To retrieve information about relative field number *n*, code:

```
&NDBINFO dbname NUMBER=n
```

In both cases, information is returned in several *user NCL variables*, prefixed by &NDBFLD. Several useful variables are listed below. See the description of the &NDBINFO verb in the *Network Control Language Reference* for the full list of returned variables.

&NDBFLDNAME
Contains the name of the field (useful when retrieving by relative number)

&NDBFLDFMT
Contains the format of the field, for example, CHAR

&NDBFLDKEY
Contains the key option for the field, for example, UNIQUE

This information can be used in a table-driven database query and update program to edit or display data without needing to code specific details about each NDB. It is also useful for unload and reload programs.

Changing NDB NCL Processing Options

The &NDBCTL verb is used to alter processing options associated with the executing NCL process. Options that can be changed are:

- Issuing messages on error conditions. By default (MSG=YES), a message is sent to the environment the NCL procedure is running under. For a normal NCL procedure, this is the OCS window. The messages are also logged. By specifying MSG=LOG, error messages are only sent to the SOLVE log.

  By specifying MSG=NO, no error messages are issued. Only database integrity messages that are sent to monitor receivers are issued.

- Handling database error conditions. By default (ERROR=ABORT), a response code greater than 29 (in &NDBRC) causes the process to be terminated with an error message.

  By specifying ERROR=CONTINUE the process retains control, but must check &NDBRC for error responses.

  Note

  &NDBOPEN is always processed as if &NDBCTL ERROR=CONTINUE is in effect, except for response 34 (already open). The same applies to &NDBCLOSE, except for response 35 (not open).

- The format that dates can be entered (by &NDBADD/&NDBUPD, and &NDBSCAN), and returned (by &NDBGET). By default (DATEFMT=*), the user's UAMS language code, or, if no specific language code is in the user definition, the system language code, determines whether dates can be entered in DD/MM/YY (UK or DATE4), or MM/DD/YY (US or DATE5) format.

  You can specify DATEFMT values of DATE1 to DATE10, UK (=DATE4), or US (=DATE5), or * (meaning as described above).

- The need to quote values for characters, hex (FMT=HEX), hex numbers (FMT=NUM BASE=HEX) and date format data using the QUOTE operand. By specifying QUOTE=NO (the default) these data types need not be quoted and embedded blanks in NCL variables in free-form text are ignored.

  By specifying QUOTE=YES all of the previously listed data types must be quoted and embedded blanks are treated as real blanks, causing &NDBQUOTE to force-quote all non-null data.

- Tracing of the parsing of free-format text. By default (TRACE=NO), no trace messages are produced.

  By specifying TRACE=YES, a message is produced for each token in the free-form text. This message is subject to the &NDBCTL MSG= setting. The first 20 characters of each token are traced.

- Dumping a scan action table. By default (SCANDEBUG=NO), no dump is produced. By specifying SCANDEBUG=YES, a dump of the generated scan action table is produced. At completion of the scan, a second dump shows record counts.

These options allow you to easily develop applications using NDBs, and to make the procedures robust.

For example, while developing code, use MSG=YES and ERROR=ABORT. This will give useful messages and stop a procedure at the error point. From time to time, TRACE=YES can be used to track obscure syntax errors in free-form text (particularly scan expressions, and problems caused by failing to quote data containing delimiter characters).

When the code is finished, specifying MSG=NO (or MSG=LOG), and ERROR=CONTINUE allows the procedures to handle unexpected errors gracefully.

The DATEFMT= option allows you to control the input and formatting of dates. The default behavior (honoring the user or system date format) is normally the best setting. However, when unloading or reloading data, &NDBCTL DATEFMT=NO allows you to be independent of the current language settings.

Putting It All Together—Unloading/Reloading an NDB

The following examples illustrate many of the points covered in this chapter. In these examples, a simple pair of procedures is developed to logically unload and reload a database.

These procedures unload any NDB to a VSAM ESDS. The unload is in RID sequence, unless the database has a sequence key, in which case it is sequenced by that key.

Note

This procedure is described to illustrate some of the features of NDBs. It is not recommended as the best way to unload files. The command NDB UNLOAD unloads an NDB in exactly the same format as the procedure described below. The NDB ALTER command can be used to speed up a reload.

Defining an Unload File

A file to hold the unloaded data is needed. This example uses a VSAM ESDS. The following definition could be used:

```
DEFINE CLUSTER (NAME(NDB.UNLOAD) -
   NIXD                          -
   RECORSIZE(40 500)             -
   CISZ(4096))
```

This dataset must be allocated to Management Services:

```
ALLOC DD=UNLOAD DSN=NDB.UNLOAD
```

The dataset must be made available to NCL, via the UDBCTL statement. When an empty ESDS is opened, a dummy record is written. Our reload program must skip this record.

```
UDBCTL OPEN=UNLOAD ID=* BUFND=5
```

The unload can now be performed.

Opening the Database and Output Unload File

The unload procedure must now open the database, and the destination unload file. Assume that the procedure is invoked from OCS with the name of the NDB and the name of the unload file:

```
$NDBUNLD dbname esdsname
```

Open the database in exclusive mode, to prevent other users updating it while it is being unloaded:

```
&DBNAME = &1  -* save db name to unload
&UNNAME = &2  -* save esds name
&NDBOPEN &DBNAME EXCLUSIVE
&IF &NDBRC NE 0  &THEN &GOTO .OPENERROR -* unable to open
&FILE OPEN &UNNAME
&IF &FILERC NE 8 &THEN &GOTO .OPENERROR  -* cant add recs
```

To preserve any lower-case data, issue &CONTROL NOUCASE:

```
&CONTROL NOUCASE  -* don't fold lower case data
```

Unloading Database Level Information

The first step is to write information about the database. You should write a record identifying this as an NDB unload dataset, and then write the information returned from an &NDBINFO DB. Use a record type of 01 for the header, and 02 for the database information:

```
&FILE ADD 01 NDB UNLOAD OF &DBNAME &DATE7 &TIME
&NDBINFO &DBNAME DB
&FILE ADD 02 &NDBDBNAME&NDBDBVKL &NDBDBVRL +
         &NDBDBNFLDS&NDBDBNRECS&NDBDBNRID
```

This information will be used by the reload program to verify the key and records lengths of the destination NDB.

Obtaining and Unloading Field Level Information

To unload and reload the data, you need to know the names of the fields you are unloading, and their attributes. Use &NDBINFO NUMBER= to extract field level information. This information (record type 10) will be written, and vartable built for use when processing data records.

To allow the reload program to easily perform field-level deletion, and so on, you should also write the relative field number as well as the field name when you unload.

Before reading the field information, set up parameters for the sequence that you will define later, defaulting it to RID. If you encounter a sequence key definition, alter the sequence parameter to FIELD=*fieldname*.

At the end of the field definitions, write a record type 11 to indicate the end:

```
&VARTABLE ALLOC ID=FTAB KEYFMT=NUM DATA=1
&SEQBY = RID
&FNUM = 0
&DOUNTIL &FNUM EQ &NDBDBNFLDS
   &FNUM = &FNUM + 1
   &NDBINFO &DBNAME NUMBER=&FNUM
   &VARTABLE ADD ID=FTAB KEY=FNUM FIELDS=DATA  +
         VARS=NDBFLDNAME
   &IF &NDBFLDKEY = SEQUENCE &THEN &SEQBY =    +
         FIELD=&NDBFLDNAME
   &FILE ADD 10 &FNUM +
      &NDBFLDNAME&NDBFLDFMT&NDBFLDKEY   +
      &NDBFLDNULLF&NDBFLDNULLV&NDBFLDUPD   +
      &NDBFLDCAPS&NDBFLDMAXL&NDBFLDDESC   +
      &NDBFLDUSER1&NDBFLDUSER2&NDBFLDUSER3 +
      &NDBFLDUSER4
   &LOOPCTL 1000
&DOEND
&FILE ADD 11
```

Building a Format for Reading Data

To efficiently read the data, you predefine a format that contains all fields. However, you rename each field to F*nnnn* on retrieval. This prevents any clashes with the control fields, if, for example, the database contains a field called I. You also ask that null fields not be returned, to save processing overheads.

You then loop through the vartable built above, to build a format using START/DATA/END:

```
&I = 1
&NDBFMT &DBNAME DEFINE FORMAT=UFMT START
&NDBFMT &DBNAME DATA FIELDS
&DOWHILE &I LE &FNUM
   &VARTABLE GET ID=FTAB KEY=I FIELDS=DATA VARS=FNAME
   &NDBFMT &DBNAME DATA (F&I = &FNAME NULLFIELD=NORETURN)
   &I = &I + 1
   &LOOPCTL 1000
&DOEND
&NDBFMT &DBNAME END
```

Defining the Sequence for Reading

The sequential retrieval path must be defined. The &NDBSEQ verb is used to accomplish this:

```
&NDBSEQ &DBNAME DEFINE SEQ=USEQ &SEQBY
```

The type of sequence, RID or FIELD=sequence key, was set when you read the field definitions.

Unloading the Data

You are now in a position to unload all the data in the NDB.

First, write a header record for the data (type 20).

Write each record out as follows:

- A header record, containing the RID and the number of non-null fields in this record. (The RID is just for information, as the reload assigns new RIDs). (Type 21).
- *n* field records, one for each non-null field in the NDB data record (type 22). These records will contain the relative field number, field name, and field value.
- A trailer record, indicating the complete NDB data record has been written (type 23). You include the number of non-null fields in the record.

At the end of all data records, a type 29 record will indicate the end of the data. This record contains the count of the number of logical records unloaded.

To protect the unload from the current language (and therefore date) settings, issue an &NDBCTL DATEFMT=NO to force dates to be returned in YYMMDD format:

```
&NDBCTL DATEFMT=NO
```

The data unload code is as follows:

```
&FILE ADD 20                      -* header for data portion
&NRECS = 0                        -* num records unloaded
&NDBGET &DBNAME SEQ=USEQ FORMAT=UFMT MODFLD=YES
-* read a record, return the fields modified in ZMODFLD,
-* ZVARCNT
&DOWHILE &NDBRC = 0                  -* till eof
   &NRECS = &NRECS + 1               -* 1 more record
   &FILE ADD 21 &NDBRID &ZVARCNT     -* record header
   &I = 1                            -* field index counter
   &DOWHILE &I LE &ZVARCNT           -* for all defined
                                     -* fields
      &FX = &SUBSTR &ZMODFLD 2       -* get unique field
                                     -* number
      &VARTABLE GET ID=FTAB +        -* and associated
                                     -* field name
         KEY=FX +
         FIELDS=DATA +
         VARS=FNAME
      &FILE ADD 22 &FX &FNAME &FX    -* write field
                                     -* num/name/value
      &I = &I + 1                    -* next field
      &LOOPCTL 1000                  -* prevent blowup
   &DOEND                            -* end all fields
   &FILE ADD 23 &ZVARCNT             -* end record
   &NDBGET &DBNAME SEQ=USEQ FORMAT=UFMT MODFLD=YES
                                     -* get next
      &LOOPCTL 1000                  -* prevent blowup
&DOEND-* end record loop
&FILE ADD 29 &NRECS-* num logical records
```

This completes the unload. All that is left is to close the files:

```
&NDBCLOSE &DBNAME
&FILE CLOSE &UNNAME
```

The database can now be used by other people. The unload can be processed as necessary, for example, copied to tape.

This example procedure is in the NCL distribution library as member $NDBUNLD. The code is as shown above. See the source for further information.

Unloading Subsets Using Sparse Keys

If an NDB contains *disjoint* record types, using keys on fields that are not in all records, you can unload just one record type by using one of those keys as the unload sequence. This can be useful for extracting subsets of the database.

Reloading an NDB from an Unload File

Having unloaded an NDB, you might want to reload it, possibly to restore the NDB to a previous state, or to take it to another system.

It is assumed that the reload is to a new NDB, that has been defined with IDCAMS, allocated to Management Services, and NDB created. These operations are described fully in Chapter 17, *Netmaster Database (NDB) Administration*.

For the purposes of the example, only minimal error checking is performed. The sample procedure can be enhanced in many ways. Some of these are:

- Allow merging of the unload data into an existing NDB
- Display a field selection list and allow field attribute changes, field renaming, and field deletion
- Subset data selection, and so on
- Using LOAD MODE to greatly speed up the load. This requires use of NDB ALTER to build keys.

Opening the Database and the Input Unload File

The procedure must first open the database, and the input unload file. It is assumed that the procedure is invoked from OCS with the name of the NDB and the name of the unload file:

```
$NDBRELD dbname esdsname
```

To speed up the reload, start the NDB in DEFER mode. This mode inhibits the database manager from flushing buffers after each &NDBADD, at the expense of an unusable database if the system fails while open this way. To set DEFER mode, the database must have been opened by the UDBCTL command with the options LSR and DEFER. The following NDB command causes deferred I/O:

```
NDB START &DBNAME DEFER
```

Note

The NDB START command can be issued at any time.

Set &CONTROL NOUCASE to protect lower case data:

```
&CONTROL NOUCASE
```

Open the database in exclusive mode, to prevent other users accessing it while it is being reloaded:

```
&DBNAME = &1                        -* save db name to reload
&RLNAME = &2                                -* save esds name
&NDBOPEN &dbname EXCLUSIVE
&IF &NDBRC NE 0 &THEN &GOTO .OPENERROR    -* unable to open
&FILE OPEN &RLNAME
&IF &FILERC GT 8 &THEN &GOTO .OPENERROR    -* unable to open
```

You must now verify the input file. Read the initial load record (and ignore it), and the second record, which should be the header record:

```
&FILE GET SEQ ARGS                -* should be initld record
&IF &FILERC NE 0 &THEN &GOTO .READERR
&FILE GET SEQ ARGS                -* should be 01 ndb ....
&IF &FILERC NE 0 &THEN &GOTO .READERR
&IF &1.&2.&3.&4.  NE 01.NDB.UNLOAD.OF. +
    &THEN &GOTO .BADFILE
&INDBNAME = &5
&INULDATE = &6
&INULTIME = &7
&WRITE RELOAD FROM BACKUP OF &INDBNAME TAKEN &INULDATE +
      &INULTIME
```

Checking Database Attributes

You must now ensure that the destination database can support the reload. Check the following:

- New database VSAM key length is GE unload NDB keylength.
- New database is empty of both fields and records.
- New database is freshly created, that is, next RID is 1. (A database emptied by deleting all records, and deleting all field definitions is not suitable).

These tests could be relaxed in a full-function reload.

The code to perform the verification is as follows:

```
&NDBINFO &DBNAME DB                 -* get info about dest
&FILE GET SEQ ARGS                  -* read next record
                                    -* (02)
&IF .&1 NE .02 &THEN &GOTO .READERR -* validate
&IF &NDBDBVKL LT &3 &THEN &DO       -* dest keylen short
   &WRITE destination keylength short, load aborted
   &END 16
&DOEND
&IF &NDBDBNFLDS NE 0 &THEN &DO      -* dest db has fields
   &WRITE destination database has fields defined, +
      load aborted
   &END 16
&DOEND
&IF &NDBDBNRECS NE 0 &THEN &DO      -* dest db has records
   &WRITE destination database has records present, +
      load aborted
   &END 16
&DOEND
&IF &NDBDBNRID  NE 0 &THEN &DO      -* dest not just
                                    -* created
   &WRITE destination database not freshly created, +
      load aborted
   &END 16
&DOEND
```

Building Field Definitions

The next records in the unload file are the field definition records. Read them and build field definitions in the new NDB. Note that &10 is not used, as it contains the field maximum length.

```
&FILE GET SEQ ARGS                  -* read next record
&DOWHILE .&1 EQ .10                 -* while it's a
                                    -* field rec
      -* see note below             -* possible changes
   &Q11 = &NDBQUOTE &11             -* protect desc
   &Q12 = &NDBQUOTE &12             -* protect user1
   &Q13 = &NDBQUOTE &13             -* protect user2
   &Q14 = &NDBQUOTE &14             -* protect user3
   &Q15 = &NDBQUOTE &15             -* protect user4
   &NDBDEF &DBNAME ADD  ( +         -* add the field
      &3+                           -* fieldname
      FMT= &4 +                     -* format
      KEY= &5 +                     -* key option
      NULLFIELD = &6 +              -* nullfield option
      NULLVALUE = &7 +              -* nullvalue option
      UPDATE= &8 +                  -* Update option
      CAPS = &9 +                   -* caps option
      DESC= &Q11 +                  -* description
      USER1= &Q12 +                 -* user1
      USER2= &Q13 +                 -* user2
      USER3= &Q14 +                 -* user3
      USER4= &Q15 +                 -* user4
      )                             -* end stmt
      &FILE GET SEQ ARGS            -* get next
      &LOOPCTL 1000                 -* prevent blowups
&DOEND                              -* till end rec 10s
&IF .&1 NE .11 &THEN &GOTO .READERR-* unexpected record
```

Note

At this point, changes to field attributes could be processed. For example:

```
&IF .&3 = .SURNAME &THEN &5 = UNIQUE
&IF .&3 = .DOB &THEN &5 = YES
```

Also, the relative field number used during unload is available in &2. This can be used during data reloading to alter field values, and so on.

At the end of this step, the new database has all the fields defined.

Loading the Data

You can now reload the data records. Each NDB record is represented by one type 21 record, *n* type 22 records, one for each non-null field, and one type 23 record.

Use &NDBADD START/DATA/END to load the records:

```
&FILE GET SEQ ARGS                    -* should be type 20
&IF .&1 NE .20 &THEN &GOTO .READERR -* bad
&NLOAD = 0                            -* num recs loaded
&OK = YES                             -* flag
&DOUNTIL &OK = NO                     -* rest of DB
   &FILE GET SEQ ARGS                 -* get a record
   &IF .&1 = .22 &THEN &DO            -* field (most common)
      -* see note (1)                 -* poss changes to
                                      -* data
      &NDBADD &DBNAME DATA +
      &3 = &Q4                        -* add the data
   &DOEND
   &ELSE &IF .&1 = .21 &THEN &DO      -* start of record
      &NDBADD &DBNAME START           -* start add of record
   &DOEND
   &ELSE &IF .&1 = .23 &THEN &DO      -* end record
      &NDBADD &DBNAME END             -* add the record
      &NLOAD = &NLOAD + 1             -* bump num recs
   &DOEND
   &ELSE &OK = NO                     -* otherwise exit
                                      -* until
   &LOOPCTL 1000                      -* prevent blowups
&DOEND
&IF .&1 NE .29 &THEN &GOTO .READERR -* unrecognized record
&IF &2 NE &NLOAD &THEN &GOTO .CNTERR
                                    -* record cnt mismatch
```

Note

Special code, to alter or skip certain fields for example, can be added here.

This completes the reload. All that is left is to close the files:

```
&NDBCLOSE &DBNAME
&FILE CLOSE &RLNAME
```

To flush the database buffers, and protect the reload, re-issue the NDB START with the NODEFER option:

```
NDB START &DBNAME NODEFER
```

This example procedure is in the NCL distribution library as member $NDBRELD. The code is basically as shown above. See the source for further information.

19

Using &NDBSCAN Statements

This chapter describes the use of the &NDBSCAN NCL statement. &NDBSCAN supports easy searching of databases. Examples are included for clarification.

This chapter discusses the following topics:

- Scan Processing
- Controlling &NDBSCAN Resource Usage
- Scan Expressions
- SQL-like Operators
- Efficient Use of &NDBSCAN
- Examples of Using &NDBSCAN

Scan Processing

Complete details of the steps taken to process an &NDBSCAN statement are included in the &NDBSCAN verb description in the *Network Control Language Reference*. A summary is presented here:

1. The scan request is parsed, and an action table built. The action table is then optimized.
2. The action table is processed and, for each action that can be processed using keys, the key records in the database are used to build intermediate results.
3. If part of the request could not be processed using keys, the final result list (from step 2) is processed by reading records, and each record is validated against the scan criteria.

Note

If none of the criteria could be processed using keys, the entire database is scanned.

4. If a sort was requested, the sort keys are extracted from passing records, and an internal sort is performed.
5. The final result list is built. This is actually done during steps 3 or 4 as part of its processing.

This might seem complex, but no knowledge of the internal processing is necessary to use &NDBSCAN. However, when performance is an issue, use of keyed fields becomes important.

Displaying the Generated Scan Action Table

The &NDBCTL statement provides an operand that allows you to obtain a display of the generated scan action table, both before the scan is processed, and after. This scan debug information can be quite useful for determining why a scan request does not return the expected records.

To obtain this display, code the following statement prior to the &NDBSCAN request:

```
&NDBCTL SCANDEBUG=YES
```

&NDBCTL MSG=YES or MSG=LOG must be in effect, otherwise no messages are returned.

The first part of the displayed table represents the parsed scan request. Each relation in the request, for example, X = Y, generates a line in the table. All generic or range field names are expanded to the full list, and actions to combine previous results, using AND, OR, or NOT, are shown. The text of the &NDBSCAN request is also displayed.

The second part of the table, produced after the scan completes, shows how many records passed each phase, and whether a scan of the actual data was required.

A sample scan debug display is shown in Figure 19-1.

Processing Scan Results

An &NDBSCAN statement can generate a list of the records that pass the supplied criteria. The list is optional, and, if it is not needed, just the number of records and the RID of the first passing record can be returned.

This list is treated like a *sequence,* defined by an &NDBSEQ statement. Records can be read using the &NDBGET statement, using SEQUENCE=*name*, where *name* is the same name as specified on the &NDBSCAN SEQUENCE=*name*.

The list can be read forward or backward, and records can be skipped, as described in Chapter 21, *&NDB Verbs, Built-in Functions, and System Variables.*

Figure 19-1. Sample SCAN DEBUG Output

```
N87K01 DEBUG DUMP OF OPTIMIZED SCAN REQUEST FOR NDB: NDB1 IID: 00000F7B.
N87K02 SEQUENCE ID  SRT KEP EXE RTD RTP OPTM   IOLIM  TIMELIM  STGLIM RECLIM
N87K03 S1           YES YES YES NO  NO  YES    10000   180.00     100    5000
N87K04 TEXT:   name = 'fred5','fred27' generic & long1 contains '
N87K05 (CONT): 72','4'
N87K06 SORT:    1 NAME         (*,*,A)
N87K06          2 ADDR         (2,5,D)
N87K10 IID  CORR-ID  PRNT TYPE     LINE COR  PRTY
N87K11 .PRI *        -    PRIMARY     1 NO  51297
N87K12 LINE NEXT TYP  PRTY S/IG RELATION INFORMATION
N87K13 .PRI    1
N87K14    1    - AND 51297    2
N87K14    2    3 REL 51297    - FIELD    NAME         =            ANY
N87K15                          VALUES   FRED5
N87K15                                   FRED27           GENERIC
N87K14    3    - REL 90000    - FIELD    LONG1        CONTAINS     ANY
N87K15                          VALUES   72
N87K15                                   4
N87K49 *END*
N87K31 DEBUG DUMP OF PROCESSED SCAN REQUEST FOR NDB: NDB1 IID: 00000F7B.
N87K32  RSP RECORDS     I/O   TIME  MAXSTG
N87K33    0       2      26   0.17    6048
N87K34 IID  K-EXE   I/O-K TIME-K R-EXE   I/O-R TIME-R    RSTG RECORDS UNQVALS
N87K35 .PRI     1      14   0.07     1      12   0.10     320       2       -
N87K35 .SRT             -      -             -   0.00    2028       2       -
N87K36 LINE |----- KEY  PROCESS -----| |-------- REC SCAN PROCESS -------|
N87K37      RESULT    SCAN       COUNT       IN     PASS     FAIL NULL-FLD
N87K38        TIME    I/O BMENT  BMSTG
N87K39 .PRI
N87K40    1 PART-FILE Y             12       12        2       10
N87K41        0.00      0     2   2016
N87K40    2 PART-FILE N             12       12       12
N87K41        0.07     14     2   2016
N87K40    3 ALL-FILE  Y                      12        2       10
N87K41        0.00      0     0      0
```

The order of records returned is *undefined*, unless SORT=*expression* was specified on the &NDBSCAN statement. This is to allow the scan request to be optimized. If the scan was sorted, the records are ordered based on the nominated sort fields, when retrieving FWD.

A sorted scan list that has exactly one full-field sort key, can be repositioned by &NDBSEQ RESET REPOS, and, as an extra option available *only* to sequences built by &NDBSCAN, can be repositioned to the record having a specific RID.

The &NDBSCAN statement allows you to specify KEEP=YES, to prevent the scan result list being deleted when an EOF response is returned by &NDBGET.

There are some important differences between sequences defined by &NDBSEQ, and sequences built by &NDBSCAN:

- &NDBSEQ-defined sequences do not have an in-storage list of records associated with them. Thus, skipping forward and backward automatically takes into account record deletions. For example, if you are reading by RID, and you are positioned on RID 10, a GET FWD SKIP=5 returns RID 15, assuming RIDS 11-14 all exist. If, while positioned on RID 15, RIDs 12 and 14 are deleted, a GET BWD SKIP=5 *does not* position you on RID 10, but, rather, RID 8 (assuming 8 and 9 exist).

  &NDBSCAN-built sequences are represented as an in-storage list of RIDs (the actual order depending on SORT, and so on). If the list is ordered on RID, the example above repositions from 10 to 15, and back to 10, as SKIP=*n* instructs &NDBGET to skip over *n*-1 entries in the list. Only when the target RID has been deleted, does &NDBGET proceed to the next RID in the list, until a non-deleted record is found.

- &NDBSEQ-defined sequences each take a small, fixed amount of storage to hold information about the sequence.

  &NDBSCAN-built sequences take storage proportional to the number of matching records to hold the list. Thus, you should try to minimize the number of concurrent scan sequences in use, particularly if using &NDBSCAN in an EASINET procedure.

Controlling &NDBSCAN Resource Usage

Since &NDBSCAN can perform large amounts of I/O, or use large amounts of storage, particularly when sorting, there are four limits that prevent any one scan request from tying up excessive resources. They are:

- Logical VSAM I/O limit
- Working storage limit
- Elapsed time limit
- Passing records limit

These limits are specified using the SYSPARMS command and are described in Chapter 20, *The NDB SYSPARMS Command*. Each limit can have a *default* value, to be applied if an &NDBSCAN does not specify an overriding value, and a *maximum* limit, that is always used to constrain the maximum value which can be coded in any &NDBSCAN.

The &NDBSCAN statement allows you to specify overriding values for any of the limits. These values replace the SYSPARM-specified defaults for that scan. If any supplied value exceeds the SYSPARM-specified maximum, the SYSPARM maximum is used.

An &NDBSCAN request that exceeds one of these limits is terminated, and a response code, indicating which limit was exceeded, is returned. The scan debug display indicates the limit values assigned to the scan request.

Scan Expressions

A scan request contains a free form text *scan expression*. This expression contains:

- Fields - for example, SURNAME, DOB
- Values - for example, SMITH, 04/12/58
- Operators - for example, =, NE
- Connectors - for example, AND, OR

Parentheses can be used to group parts of the expression.

The expression can be spread across several NCL statements, using the START/DATA/END syntax, as per the &NDBADD, &NDBUPD, and &NDBFMT statements. This syntax also allows construction of the scan expression. This is especially useful in table-driven systems, for example, the Management Services features table.

The expression consists of a number of *scan-tests*, connected by AND, OR, NOT, and parentheses. Each test can:

- Test a field, or list of fields (including generic and range field names), against a value, list of values, or generic value(s) *or* against other fields. This is called a field to field compare.
- The comparison can use the standard relational operators, for example, EQ, NE, GT, and =, >, *as well as*, special operators:
  - PRESENT—to test for presence
  - ABSENT—to test for absence
  - LIKE—to perform a pattern match
  - CONTAINS—to test, for character fields, for one field containing a value, or other field value
- SQL-like capabilities exist that allow nested scans, where field values from the inner scans are used as input to the tests in the outer scan(s).

The following are examples of valid scan expressions:

```
SURNAME = SMITH AND DOB LT 600101
DESC* CONTAINS MVS
DATECLOSED ABSENT OR DATECLOSED  DATEOPEN PLUS 5
NAME LIKE 'J% SMITH%'
SEX = 'F' AND (STATUS = 'SINGLE' | STATUS = 'DIVORCED')
SEX = 'F' AND STATUS = ANY 'SINGLE', 'DIVORCED'
```

The last two expressions are equivalent. The parentheses are required in the first of these to bind the OR (|) tests together, as AND takes precedence over OR.

Reserved Words

The syntax for a *scan expression* (as documented in the &NDBSCAN verb description in the *Network Control Language Reference*), shows words such as ALL, ANY, FIELDS, and VALUES. These words are used to indicate options in scan tests. Although it might appear that these are reserved words, and thus cannot be used as field names, or unquoted field values, this is not the case.

There are no reserved words in the syntax for a scan expression.

All keywords are resolved by *context*. That is, if a certain keyword is allowed at a point in the expression, the presence of that keyword at that point in an expression is regarded as being that keyword. At any other point, that keyword is regarded as a name, value, and so on.

For example, the following are valid scan expressions. Keywords are shown in bold typeface. (Assume that field names and so on are defined.)

```
FIELD ALL = VALUE ANY
ANY FIELDS VALUE, FIELD, ALL CONTAINS ALL VALUES FIELD, +
VALUE, ANY
```

This can cause some confusion. Obviously, fields names of FIELD, ALL, and so on, are not recommended.

When generating scan expressions dynamically, it is always a good idea to insert all the optional keywords, to prevent a syntax error when, for example, a search value of FIELDS is provided. For example:

```
&NDBSCAN ...  DATA FIELD1 = &SCHVALUE
```

If the variable &SCHVALUE contained the characters FIELDS, without the quotes, the scan would fail with a syntax error, as the keyword FIELDS is not followed by a field name.

To prevent this, the example above could be coded:

```
&NDBSCAN ...  DATA FIELD1 = VALUE &SCHVALUE
```

Null Fields

Null (that is, *not present*) fields are handled in a special way by &NDBSCAN:

- A null field *never* matches a field to value test. A record with field SURNAME *not present* is not *equal* to a supplied value, nor is it *not equal* to a supplied value. This 'null result' carries through the scan action table.
- A null field can only be selected using the ABSENT or IS NULL operators.

This handling of null fields by &NDBSCAN is consistent with other &NDB verbs. Thus, *disjoint* record types work as expected.

Field to Field Comparisons

As mentioned above, you can compare one field with another in a record. A good example of this is finding all records with WITHDRAWALS GT BALANCE.

Note

Field to field comparisons involve scanning records. Always try to have other (keyed) criteria ANDed with these criteria.

When performing field to field comparisons on numeric, float or date fields, an adjustment amount (taken as a number of days, for date format data), can be specified. For example, the following statement matches all records where the field DATECLOSED was greater than the field DATEOPEN plus 10 days:

```
DATECLOSED GT DATEOPENED PLUS 10
```

The amount must be an integer; floating point numbers are not supported.

When performing nested scans (sub-selects), you can perform a field-to-field comparison when one of the fields is the current value in an outer scan. This is called a *correlated* query.

Using &NDBQUOTE to Protect Special Characters

As mentioned in the previous chapter, the &NDBQUOTE built-in function should be used to protect data values whenever there is the possibility of delimiter characters (for example, =, &) being present. This recommendation also applies to arguments supplied in a scan expression.

For example, to search for a value of A=B, use these statements:

```
&SCHARG = &NDBQUOTE A=B
&NDBSCAN MYNDB SEQ=S1 DATA FIELD1 = &SCHARG
```

Failure to quote the value results in a syntax error because the equal sign is treated as a delimiter.

Searching for Lower Case Data

An NDB can store character data in lower case. When a character field is defined, CAPS=YES is assumed, which means that both the stored data, *and the key* are folded to upper case. There are two other options:

CAPS=NO

The data is left as is, that is, lower case data is left lower case, including in the key. For example, ABC and *abc* are regarded as different values.

CAPS=SEARCH

The data is left as is, that is, lower case data is left lower case. If the data is keyed, the key *is folded to upper case*. Thus, the values ABC, and *abc* are regarded the same when building a key, but, when data is returned, the lower case version is retained.

For &NDBSCAN, CAPS=SEARCH also applies to *non-keyed* fields or processing. This means that, when reading data records, fields defined with CAPS=SEARCH are folded to upper case when they are examined.

Supplied search arguments are retained in lower case (or as entered), and are upper cased as required (that is, when comparing to CAPS=YES or CAPS=SEARCH fields).

Note

&CONTROL NOUCASE must be in effect to preserve lower case information supplied on an &NDBSCAN statement.

Using CONTAINS

The CONTAINS operator is extremely useful when scanning text in a database. When combined with generic or range field names, a single expression can perform quite sophisticated lookups.

Some notes about the use of CONTAINS:

- CONTAINS involves scanning records. Always try to have other criteria ANDed with CONTAINS, to reduce the number of records that must be scanned.
- The field(s) on the left of CONTAINS are padded at each end with one blank, for the purposes of searching. This allows words to be searched for, by providing blanks around your search arguments, without worrying that a word at the front or back of a field will not be matched. For example, a field value of A SENTENCE OF WORDS is regarded as A SENTENCE OF WORDS when processed by CONTAINS. Thus, searching for the value A will succeed, even though the data does not contain a blank in front of the A.

- The ANY and ALL keywords allow some requirements for text to be adjacent to be tested. For example the following statement only matches records where *any one* of the fields prefixed by DESC contains *both* WORD1 and WORD2:

  ```
  ANY FIELD DESC* CONTAINS ALL WORD1, WORD2.
  ```

  If each DESC... field held a sentence, this is equivalent to asking for a sentence containing both words.

  Refer also to the LIKE operator, discussed below.

- Fields *in the same record* can contain the search argument(s). Thus, it is possible to find all records where a given character field in the records is contained within another character field.

Using LIKE

The LIKE operator allows pattern matching to be performed. The NOT LIKE operation does the same, but matches records without the pattern.

The argument string for LIKE can contain any characters except for two special characters: **_** (underscore) and **%** (percent). These characters only match themselves (the CAPS=NO/SEARCH rules are honored).

The special characters obey the following rules:

- Underscore (_) matches any one character, but there must be a character in the data at this position.
- Percent (%) matches zero or more characters.

These special characters can be used as many times as required in a search string. Some examples follow:

ABC	Matches only records with ABC in the field
ABC%	Matches on values starting with ABC
ABC_%	As above, but at least one character must follow ABC
%XYZ	Matches fields ending in XYZ
%mmm%	Matches fields with mmm somewhere in them
%FRED%_%BLOGGS%	Matches a field containing the strings FRED and BLOGGS in that order, with at least one character between them
_ _ _	Matches a field exactly 3 characters long
_ _XY_%	Matches a field at least 5 characters long, with XY in columns 3 and 4

It is evident that LIKE is a powerful operator and consequently can use significant CPU resources.

The first two examples above can also be done using keys. LIKE attempts to use keys to fully (as above) or partially (whenever a LIKE argument has non-special leading characters) determine matching records.

Using the Results of a Previous &NDBSCAN

The syntax, SEQUENCE *seqname* allows a scan to use as part of its input, the result list from a previous scan. This avoids having to re-evaluate an entire previous scan, just to add some more criteria to it.

In interactive applications, this can be very powerful. For example, after performing a scan based on user input, a panel can be displayed informing the user of the number of hits. One option the user can have is to add extra criteria, and see the result of the extra criteria, *as applied to the current result list*. The display, extra criteria, and so on, cycle could be repeated. By using the previous scan as input, large amounts of system resources can be saved. Also, as the user ascends the nested levels of display, previous results are still valid.

This option is also useful when a scan expression that is too complex to handle in one statement is needed. It can be broken into parts, and the results combined.

For example:

```
&NDBSCAN MYNDB SEQ=S1 DATA SURNAME = 'SMITH'
&NDBSCAN MYNDB SEQ=S2 DATA DOB LT 600101
&NDBSCAN MYNDB SEQ=RESULTS DATA SEQUENCE S1 AND +
    SEQUENCE S2
```

Note

A scan expression can just include a previous scan result. This is useful when you want to re-sort a scan result without rebuilding as list. For example:

```
&NDBSCAN DB2 SEQ=S1 SORT=FIELD1 DATA ...scan expr
&NDBSCAN DB2 SEQ=S2 SORT=FIELD2 DATA SEQUENCE S1
```

SQL-like Operators

Scans can perform some SQL-like functions:

- The IS [NOT] NULL operators provide equivalent functionality to the PRESENT and ABSENT operators, but are SQL-compatible.
- The [NOT] BETWEEN operators are equivalent to the =value:value and p=value:value operators, and are SQL-compatible.
- The [NOT} IN operators are equivalent to the [p]= value-list operators, and are SQL-compatible.
- The SELECT clause allows you to perform sub-selects, where a list of values derived from a set of records matching some criteria can be used as input to an outer-level scan.

  The EXISTS operator uses the fact that at least one record exists in the inner select/scan to determine truth or falsity.

  A correlated select is possible. This causes an inner scan to be re-executed once for each other outer record that passed part of a scan. The outer record field values are used in the inner scan as comparison values.

Efficient Use of &NDBSCAN

&NDBSCAN can be misused. The scan limits discussed previously are designed to prevent excessive use of resources. When designing applications that use &NDBSCAN, the following should be taken into account:

- Although you can search on any field, use of keyed fields, connected by AND, at the outer level of the scan expression, greatly reduces I/O and elapsed time. Even one keyed field can have a significant impact. For example, the following statement reads every record on the database, if field NAME has no key:

  ```
  NAME = 'SMITH' AND ADDRESS CONTAINS 'STREET'
  ```

  The CONTAINS is always evaluated by reading records. If, however, field NAME is keyed, only records with that field equal to SMITH are read, to process the CONTAINS criteria.
- PRESENT and ABSENT use keys if possible. PRESENT makes better use of keys. LIKE uses keys if there are non-special ('_', '%') characters at the front of the pattern. It might still need to examine the actual record.

- The CONTAINS operator *always* requires records to be read to evaluate success or failure. If you are performing many keyword searches of text using CONTAINS, consider storing the words of the text field as individual, keyed fields. These can then be searched for directly. For example the following statement scans the entire database:

  ```
  DESCRIPTION CONTAINS 'WORD'
  ```

  If, however, as well as field DESCRIPTION, you had fields DESCWD01-DESCWD10, all keyed, containing the individual words of the field DESCRIPTION (&PARSE could be used), you could code the following statement which could use keys:

  ```
  ANY DESCWD* = 'WORD'
  ```

- For interactive applications, encourage the use of scan criteria which reduce the number of hits. If a given scan returns more hits than could be validly processed by the user, display a message requesting more and/or more selective criteria. The scan option to use the results of a previous scan can also be useful in this case.

- Avoid the use of OR at the highest level of the scan expression, if either side of the OR involves a non-keyed field, or PRESENT, ABSENT, or CONTAINS—a full database scan results.

- Only use SORT when absolutely necessary. Sorting will normally involve reading all the records that pass the criteria, extracting the sort field, performing an in-storage sort, and then building the result list. The sort key extraction is done concurrently with final record scanning if non-keyed, and so on, criteria are to be processed.

  This processing can consume large amounts of I/O, as well as storage for the sort keys, and can take significant time.

- Nested scans (SELECT causes) can use large amounts of storage to store inner results.

  Correlated scans can perform multiple passes over the database. This is because an inner scan can be called once for every record in the NDB, and it too can read the NDB, or a large portion of it.

- If you only want to know whether any records at all have, or have not, passed the scan criteria, and you are not interested in the exact number or specific IDs of any such records, then use parameter RECLIMIT=1. &NDBSCAN RECLIMIT=1 does not set the &NDBRID variable.

If sorting is unavoidable, for example, because you need to be able to reposition in the output sequence, always try to minimize the number of records passing the scan.

Examples of Using &NDBSCAN

Some examples of the use of &NDBSCAN appear below. Each example is followed by a description of the processing.

1. Find all open problems that have not been assigned to anyone for resolution:

   ```
   &NDBSCAN PROBDB SEQ=S1 START
   &NDBSCAN PROBDB DATA   PROBSTAT = 'O' AND PROBASSGND +
   ABSENT
   &NDBSCAN PROBDB END
   ```

   The scan expression has been coded separately. This makes any future additions to the expression easy.

   PROBSTAT=O matches all open problems. PROBASSGND ABSENT matches all record where the PROBASSGND field has not been set.

2. Look up a library database for all books co-authored by AHO:

   ```
   &NDBSCAN LIBDB SEQ=S2 START
   &NDBSCAN LIBDB DATA   AUTHOR* CONTAINS ' AHO '
   &NDBSCAN LIBDD DATA   AND AUTHOR2 PRESENT
   &NDBSCAN LIBDD END
   ```

   AUTHOR* CONTAINS ‘ AHO ’ matches all records where any field name AUTHOR... contains the string ‘ AHO ’, and AUTHOR2 PRESENT matches those records where the field AUTHOR2 is actually present in the record. That is, in this example, we assume AUTHOR1, AUTHOR2, ... contain the author names.

3. A more efficient version of (2):&NDBSCAN LIBDB2 SEQ=S3 START
   &NDBSCAN LIBDB2 DATA AUTHORSN* = 'AHO' AND NAUTHORS + GT 1
   &NDBSCAN LIBDB2 END

   Here, it is assumed that the author’s surname has been keyed as field AUTHORSN*nn*, and that the number of authors is available in a field called NAUTHORS.

4. Look up the library database for all books containing the keywords SORTING and SEARCHING in their title. If you are familiar with SSAs in IMS or DL/I, you will recognize a similarity to the independent AND available when using secondary indexes:

   ```
   &NDBSCAN LIBDB SEQ=S4 START
   &NDBSCAN LIBDB DATA TITLEKWD* = SORTING AND +
   TITLEKWD* = SEARCHING
   &NDBSCAN LIBDB END
   ```

   Here, it is assumed that the title is broken into words, and each word is stored in fields named TITLEKWD*nn*. Thus, we can perform a fast key search to find a list of all books with each keyword, and then the lists are ANDed, to obtain the intersection.

20

The NDB SYSPARMS Command

The NDB SYSPARMS command can be used to change many system parameters. Its use is usually restricted to system administrators.

There are a number of commands executable from an OCS window, or from an NCL procedure running in either fullscreen or non-fullscreen mode. These commands are described in the *Management Services Command Reference*.

This chapter discusses the following topic:

- Format of the SYSPARMS Command

Format of the SYSPARMS Command

The SYSPARMS command is presented in the following format:

- The precise syntax for the SYSPARMS command is displayed in a box. The full command name appears in capital letters at the left of the box. Below the command name in brackets is the default minimum authority level required to execute the command. The default authority level can be modified by the SYSPARMS CMDAUTH command.
- The operands that are applicable to the command appear to the right of the command name.

```
SYSPARMS     [ NDBSCANO={ YES | NO } ]
             [ NDBSUBMN=n ]

(4)
```

Syntax Guidelines

The syntax used follows these guidelines:

UPPERCASE Characters

Command names or operands consisting of uppercase characters must be entered as shown but can be entered in upper or lower case. Command names (but not operands) can be truncated to the minimum string that makes them unique.

***Italic* Characters**

Italic characters are variables that show the kind of information, rather than the exact information, that must be supplied. The actual entry replaces the lowercase description.

Underscored Values

An underscored value indicates the default value that is assumed for an operand if that operand is not specified in the command.

Braces { }

Braces are used to indicate the available options for a required operand. One of the alternatives described must be selected. Do not include the braces when entering the desired option.

Square Brackets [ ]

When square brackets enclose an operand, they indicate an optional specification. This includes any accompanying equal signs. Do not include square brackets when entering the specification.

Or-sign |

The or-sign is used to separate options for an optional or mandatory specification. If a group of options is enclosed by square brackets, and the individual options are separated by or-signs, none of the options in the group has to be specified. If none of these operands are coded, the default value is used. Default values are always underscored.

Commas and Equal Signs

Commas and equal signs must be entered as shown. When commas and equal signs appear within brackets, they are optional and are to be used only if the accompanying optional operand is used.

The SYSPARMS Command

Initializes or modifies system parameter values.

```
SYSPARMS    [ NDBDIOL=n ]
            [ NDBDRCL=n ]
            [ NDBDSTL=n ]
            [ NDBDTML=n ]
            [ NDBMIOL=n ]
            [ NDBMRCL=n ]
            [ NDBMSTL=n ]
            [ NDBMTML=n ]
            [ NDBSCANO={ YES | NO } ]
            [ NDBSUBMN=n ]
            [ NDBSUBMX=n ]
            [ NDBLOGSZ=n ]
            [ NDBOPENX={ NO | YES } ]
            [ NDBPHONX=name ]
(4)
```

Use:

The NDB SYSPARMS operands alter the operating characteristics of Netmaster databases (NDBs). The operand descriptions indicate the use of each parameter.

Operands:

NDBDIOL=*n*

Sets the default logical I/O limit to be placed on an &NDBSCAN statement that does not have the IOLIMIT= parameter specified, or has it specified as 0 or null. The system default value is 2000. The value specified can range from 0 to 1,500,000.

A logical I/O is a single request to VSAM. Do not be afraid to use high values, as, typically, it can take 10 logical I/Os to generate one physical I/O. A scan exceeding the I/O limit is terminated with a warning response.

Note

Regardless of how the scan gets an I/O limit, it is always rounded down to the NDBMIOL value if it is exceeded.

NDBDRCL=*n*

Sets the default limit for the number of records that can satisfy an &NDBSCAN statement that does not have RECORDLIMIT= specified, or has it specified as 0 or null.

The value specified can range from 0 to 1,000,000.

If the number of records passing a given scan meets or exceeds this limit the scan is terminated with a warning response.

Note

Regardless of how the scan gets a record limit, it is always rounded down to the NDBMRCL value if it is exceeded.

NDBDSTL=*n*

Sets the default limit in Kbytes, for the amount of working storage that can be used while processing an &NDBSCAN statement that does not have STGLIMIT= specified, or has it specified as 0 or null.

Scan working storage is used to hold intermediate result lists, bitmaps, and keys for sorting.

The value specified can range from 4 to 4,000.

If the amount of working storage used exceeds this value, the scan is terminated with a warning response.

Note

Regardless of how the scan gets a storage limit, it is always rounded down to the NDBMSTL value if it is exceeded.

NDBDTML=*n*

Sets the default limit in seconds, that can elapse when processing an &NDBSCAN statement that does not have TIMELIMIT= specified, or has it specified as 0 or null.

The value specified can range from 1 to 3,600.

If the elapsed time exceeds this value, the scan is terminated with a warning response.

Note

Regardless of how the scan gets a time limit, it is always rounded down to the NDBMTML value if it is exceeded.

NDBMIOL=*n*

Sets the maximum logical I/O limit allowed by an &NDBSCAN statement. This value overrides any value provided, either on the statement, or by default from NDBDIOL, if the NDBMIOL value is lower.

A logical I/O is a single request to VSAM. Do not be afraid to use high values, as, typically, it can take 10 logical I/Os to generate one physical I/O.

The value specified can range from 0 to 1,500,000.

A scan exceeding the I/O limit is terminated with a warning response.

Note

Regardless of how the scan gets an I/O limit, it is always rounded down to the NDBMIOL value if it is exceeded.

NDBMRCL=*n*

Sets the maximum limit for the number of records that can satisfy an &NDBSCAN statement. This value overrides any value provided, either on the statement, or by default from NDBDRCL, if the NDBMRCL value is lower.

The value specified can range from 0 to 1,000,000.

If the number of records passing a given scan meets or exceeds this limit the scan is terminated with a warning response.

Note

Regardless of how the scan gets a record limit, it is always rounded down to the NDBMRCL value if it is exceeded.

NDBMSTL=*n*

Sets the maximum limit in Kbytes for the amount of storage that can be used while processing an &NDBSCAN statement. This value overrides any value provided, either on the statement, or by default from NDBDSTL, if the NDBMSTL value is lower.

The value specified can range from 4 to 4,000.

If the amount of working storage used exceeds this value, the scan is terminated with a warning response.

Note

Regardless of how the scan gets a storage limit, it is always rounded down to the NDBMSTL value if it is exceeded.

NDBMTML=*n*

Sets the default limit in seconds that can elapse when processing an &NDBSCAN statement. This value overrides any value provided, either on the statement, or by default from NDBDTML, if the NDBMTML value is lower.

The value specified can range from 1 to 3,600.

If the elapsed time exceeds this value, the scan is terminated with a warning response.

Note

Regardless of how the scan gets a time limit, it is always rounded down to the NDBMTML value if it is exceeded.

NDBSCANO= {YES | NO}

Enables (YES) or disables (NO) the scan optimizer.

Note

The setting of the NDBSCANO value does not affect NDBs that are already started at the time the command was issued.

An individual NDB can override the setting of the NDBSCANO command using the OPTIMIZE operand of the NDB START command.

NDBSUBMN=*n*

Sets the minimum number of subthreads that will stay active, for any NDB, awaiting database requests that can run asynchronously (&NDBSCAN and &NDBGET requests). When a database request arrives that can be run asynchronously, the database handler starts a separate copy of itself to run that request, unless there are already NDBSUBMX subthreads running. As the subthreads run out of work, they terminate unless the NDBSUBMN limit is reached.

The system default is 3. The value can range from 0 to 20.

NDBSUBMX=*n*

Sets the maximum number of subthreads allowed. (see the description above of the NDBSUBMN operand).

The system default is 5. The value can range from 1 to 20.

NDBLOGSZ=*n*

Sets the number of VSAM logical records that will be formatted as a journaling area when an NDB is created using the NDB CREATE command. This journal area is used to provide transaction integrity across system failures. If the LOGSIZE parameter is specified on NDB CREATE, it overrides this value.

The system default is 40. The value can range from 10 to 200.

To change the journal size for that database, change this value prior to issuing an NDB CREATE command.

Journal size is influenced by the possible complexity of an add, update, or delete operation on the database, which in turn depends on the size of the data record, and the number of keys being added, and so on. See Chapter 17, *Netmaster Database (NDB) Administration*, for further details. The journal automatically extends if it is under-allocated.

NDBOPENX={NO| YES}

Controls whether the nominated NCLEX01 is called for &NDBOPEN.

NDBPHONX=*name*

Registers name of NCL phonetic exit program. See the description of the exit for more details.

Examples:

```
SYSPARMS NDBDIOL=200 NDBDRCL=200 NDBDSTL=10 NDBDTML=10
```

Sets the default limits to prevent default scans from using excessive resources.

```
SYSPARMS NDBMIOL=200000
```

Allows an &NDBSCAN to specify a large IOLIMIT value.

```
SYSPARMS NDBLOGSZ=100
```

Makes the next NDB CREATE command build a journal area of 100 maximum-length VSAM records for transaction and integrity management.

Notes:

The NDB SYSPARMS can be changed at any time, not just in NMINIT or NMREADY. This is particularly useful for the scan limits, as any change to these are immediately used for all new requests. Scan requests running at the time use the values as they were at the time the request started. The same applies to the NDBSUBMN and NDBSUBMX parameters.

The NDBLOGSZ SYSPARM can be changed prior to an NDB CREATE if required. Alternatively, use the LOGSIZE parameter on NDB CREATE.

See Also:

The &NDBSCAN verb description in the *Network Control Language Reference* manual, and the NDB CREATE command description in the *Management Services Command Reference* manual.

21

&NDB Verbs, Built-in Functions, and System Variables

This chapter provides a summary of verbs, built-in functions, and system variables used by Netmaster databases (NDBs). Detailed descriptions of each of these are in the *Network Control Language Reference* manual.

This chapter discusses the following topics:

- &NDB Verb Summary
- Built-in Function Summary
- System Variable Summary
- Free-form Syntax

&NDB Verb Summary

&NDBADD
Adds a new record to an NDB.

&NDBCTL
Sets NDB NCL processing options.

&NDBCLOSE
Signs off (disconnect) from an NDB.

&NDBDEF
Adds or deletes field definitions to/from an NDB.

&NDBDEL
Deletes a record from an NDB.

&NDBFMT
Defines a data format list for retrieval of data from an NDB.

&NDBGET
Retrieves a record from an NDB.

&NDBINFO
Retrieves information about an NDB or field definitions in the NDB.

&NDBOPEN
Signs on (connect) to an NDB.

&NDBSCAN
Finds a set of records in an NDB that matches a set of search criteria.

&NDBSEQ
Defines a sequential retrieval path into an NDB.

&NDBUPD
Updates a record in an NDB.

Built-in Function Summary

&NDBPHON
The &NDBPHON function is a phonetic conversion function which allows conversion of a character string into a phonetic key. The conversion algorithm supplied is SOUNDEX. However, a user-supplied exit can be called. The user exit can implement any approach to phonetic conversion.

The SOUNDEX algorithm is as per KNUTH (*Art of Computer Programming, Volume III*). The returned value is K*nnn* where K is the first character and *nnn* is the SOUNDEX coded value.

A sample exit, PHONEX01, shows how to write a phonetic exit. The NDBPHONX SYSPARMS supply the name of the user exit.

&NDBQUOTE
Allows an NCL procedure to automatically quote data for the &NDBADD, &NDBGET, &NDBSCAN, and &NDBUPD statements.

System Variable Summary

&NDBERRI
Provides extra information about some errors.

&NDBRC
Provides a return code indicating success or otherwise after an &NDB statement.

&NDBRID
Provides the current record ID after some &NDB statements.

&NDBSQPOS
Indicates the position of a returned record in a sequence built by &NDBSCAN.

Free-form Syntax

Several NCL statements use a special syntax, different from normal NCL syntax, to allow easy coding of data definitions and scan requests. The relevant statement descriptions indicate the part of the statement that uses the free-form syntax. The free-form part must always be coded after any fixed-form, standard NCL-syntax parameters on the same statement. The rules for this free-form syntax are:

- Blanks are only required to delimit adjacent words, except inside data values. Blanks are not required, but may be specified, around or next to special characters (listed below) that act as delimiters.

Blanks inside data values are significant, except that trailing blanks are never stored in character-format data.

Note

NCL variables with blanks in the value are regarded as a special case, and the blank is regarded as part of the data value. This is because blanks inside NCL variables are represented internally in a special way.

- The following special characters act as delimiters, unless enclosed in a quoted string. They have special meaning to the syntax:

(	Left bracket
)	Right bracket
:	Colon (meaning: range)
=	Equal sign
¬	Not sign
<	Less than sign
>	Greater than sign
&	Ampersand (meaning: AND)
|	Bar (meaning: OR)
,	Comma
	Real blank (not embedded in an NCL variable)

  Certain combinations of these characters are treated as a single token for parsing. These combinations are p =, < =, >=, and =<, =>, meaning not equal, less than or equal, greater than or equal, less than or equal, and greater than or equal, respectively.

- Values may be enclosed in quoted strings whenever the value contains a special character, or a *real* blank.

  The quotes can be single (’) or double(”). If the data value being quoted contains a single or double quote, you can quote the data with the other quote, or double up each occurrence of the quote character.

  For example, *‘It”s a quoted value’* will be regarded as the value *It’s a quoted value*.

  The &NDBQUOTE built-in function provides an easy way to automatically quote data when necessary.

  A data value can always be quoted, even date, hex, or numeric values. Quoting also prevents any possibility of the value being regarded as a keyword.

- The following words can be used instead of special characters, as an aid to clarity. If surrounded by other words, ensure at least one blank separates them.

EQ	Can be used to replace	=
NE	Can be used to replace	¬=
LT	Can be used to replace	<
GT	Can be used to replace	>
LE	Can be used to replace	=<
GE	Can be used to replace	=>
AND	Can be used to replace	&
OR	Can be used to replace	|
TO	Can be used to replace	:
NOT	Can be used to replace	¬

- Several statements support a START/DATA/END construct to allow free-form expressions to be constructed that are longer than a single NCL statement is allowed to be. These statements can be coded as:

  `&NDB`*`xxxx`* *`dbname`* `[ parameters ] [ DATA ]` *`free-form text`*

  if the free-form text can fit on one statement (with possible continuations).

  To overcome NCL statement length limitations, and also to allow the free-form text to be built piece-meal (for example, by indirect variable reference), the statements can also be coded as:

  ```
  &NDBxxxx dbname [ parameters ] START
  &NDBxxxx dbname [ DATA ]  part-of-free-form-text
  &NDBxxxx dbname END
  ```

  The free-form text can be broken anywhere a blank is valid. Any number of intermediate statements can be used to build the complete free-form text. The database is not called until the END statement is encountered.

  Any other parameters must be coded on the &NDB*xxx* START statement.

  Note

  The statements may be interspersed with other NCL statements, including statements referencing other or even the same database, and even statements building free-form text for the same database, as long as they are different statements. That is, you can be concurrently building a multi-statement add and update for the same database, but not two different adds for the same database.

  To cancel a partially built statement, use:

  `&NDB`*`xxx`* *`dbname`* `CANCEL`

  This statement is valid even if no current START/END set is being built; thus it can be used in general error routines.

A

NCL Verb and Built-in Function List

This chapter lists two special groups of entities used with NCL, verbs and built-in functions, with a brief description of their use.

A full description of each NCL verb and built-in function, and details of its use, can be found in the *Network Control Language Reference* manual.

Summary Table

Table A-1. Verb and Built-in Function Summary Table (Sheet 1 of 6)

Name	Verb or Built-in	Description
&AOMALERT	V	Generates or simulates an AOM event, MVS WTO, VM MSG, or MVS DOM, and routes it as required
&AOMCONT	V	Releases a message from an AOMPROC for delivery, passes the message to another AOMPROC for processing, or sends a copy of the current message to an ISR connected system
&AOMDEL	V	Deletes the message currently being processed by an AOMPROC
&AOMFLAG	V	Alters the value of an AOM global flag
&AOMFLAG	B	Inspects the value of an AOM global flag
&AOMGVAR	V	Alters the value of an AOM global variable
&AOMGVAR	B	Inspects the value of an AOM global variable
&AOMINIT	V	Indicates that the current procedure is to be regarded as an AOMPROC, and registers the procedure for message delivery
&AOMMIGID	B	Determines whether a migration ID is required.
&AOMMINLN	B	Accesses the text of a specific minor line of a multi-line WTO message in an AOMPROC
&AOMMINLT	B	Accesses the line type of a specific minor line of a multi-line WTO message in an AOMPROC
&AOMREAD	V	Requests that the next message be made available to an AOMPROC
&AOMREPL	V	Alters the text of a message and release the message for local delivery
&APPC	V	Provides access to LU6.2 conversations
&APPSTAT	B	Returns the current status for a VTAM application
&ASISTR	B	Assigns a multi-word string into a variable, retaining leading blanks
&ASSIGN	V	Updates lists of variables in one operation
&BOOLEXPR	B	Evaluates a boolean expression
&CALL *procedure*	V	Invokes an NCL procedure
&CALL *program*	V	Invokes a user program
&CMDLINE	V	Writes text into your OCS command input line
&CNMALERT	V	Sends a CNM record directly to CNMPROC in a local or remote NEWS system for processing
&CNMCLEAR	V	Requests that all outstanding CNM reply data solicited by this NCL user be cleared
&CNMCONT	V	Directs the current CNM record across a specific ISR link
&CNMDEL	V	Deletes a CNM record or stops ISR delivery of the record to a remote SOLVE system
&CNMPARSE	V	Requests that the MDO data supplied be parsed into user variables

Table A-1. Verb and Built-in Function Summary Table (Sheet 2 of 6)

Name	Verb or Built-in	Description
&CNMREAD	V	Requests that the next CNM record be made available to an NCL procedure
&CNMSEND	V	Requests that the data supplied be sent across the CNM interface
&CNMVECTR	V	Requests that the data supplied be vectored into user variables
&CONCAT	B	Concatenates multiple variables/constants
&CONTROL	V	Sets NCL procedure control characteristics
&DATECONV	B	Changes a date format
&DEC	B	Converts a hexadecimal number to its decimal equivalent
&DECODE	V	Decodes part or all of an MDO
&DELAY	V	Interrupts processing of a procedure for a specified period of time
&DO	V	Groups a sequence of NCL statements to form a logical program function
&DOEND	V	Signifies the logical end of a group of statements
&DOM	V	Issues an MVS DOM to cause a non-roll-delete WTO to be erased
&DOUNTIL	V	Builds a conditional loop with a test at the bottom
&DOWHILE	V	Builds a conditional loop with a test at the top
&EDB	V	Opens and closes a connection from NCL to an external database, or sets the NCL system and user variables
&ELSE	V	The code following the &ELSE verb is the alternative path after &IF, where the &IF condition is false
&ENCODE	V	Encodes all or part of an MDO
&END	V	Terminates the current nesting level
&ENDAFTER	V	Terminates the current nesting level after executing the command following the &ENDAFTER
&EXIT	V	Terminates the current nesting level
&EVENT	V	Signals an event occurrence
&FILE	V	Connects, disconnects, switches, accesses, modifies, and deletes file records
&FLUSH	V	Terminates all nesting levels within an NCL process
&FNDSTR	V	Determines whether a string occurs within one or more variables
&GOSUB	V	Branches to a subroutine within the procedure
&GOTO	V	Branches to another statement within the procedure
&HEX	B	Converts a decimal number to its hexadecimal equivalent
&HEXEXP	B	Converts a character string to its hexadecimal equivalent
&HEXPACK	V	Converts a hexadecimal string into equivalent characters
&IF	V	Tests the truth of a logical expression
&INTCLEAR	V	Clears messages queued to a dependent processing environment

Table A-1. Verb and Built-in Function Summary Table (Sheet 3 of 6)

Name	Verb or Built-in	Description
&INTCMD	V	Schedules a command to execute in the issuing process's dependent environment
&INTCONT	V	Propagates a message to the next higher processing environment
&INTREAD	V	Retrieves the next message queued from the issuing process's dependent processing environment
&INTREPL	V	Propagates a message to the next higher processing environment, and changes the message text
&INVSTR	B	Inverts a string
&LBLSTR	B	Removes leading blanks from a string
&LENGTH	B	Tells you the length of a variable or constant
&LOCK	V	Obtains or releases access to a resource
&LOGCONT	V	Resumes normal processing of a message delivered to LOGPROC
&LOGDEL	V	Deletes a log record being processed by LOGPROC
&LOGON	V	Passes control of a terminal to another application
&LOGREAD	V	Makes the next log message available to LOGPROC
&LOGREPL	V	Replaces the text of the last log message delivered to LOGPROC
&LOOPCTL	V	Sets a new runaway loop control limit
&MAICMD	V	Specifies an MAI primary command
&MAICONT	V	Sends the current datastream on to the terminal and/or the application
&MAICURSA	V	Sets up the cursor address to be sent to the application
&MAIDEL	V	Signifies that a datastream is not to be delivered
&MAIDSFMT	V	Places the entire current datastream into variables
&MAIFIND	V	Determines whether a datastream contains a given string
&MAIINKEY	V	Sets the attention key that is to be simulated in a datastream
&MAIPUT	V	Builds a datastream to be sent to the PLU (application)
&MAIREAD	V	Waits for the next datastream
&MAIREPL	V	Replaces a datastream destined for the terminal
&MAISADD	V	Adds a new session definition, based on user variables
&MAISCMD	V	Specifies an MAI session command against the current session
&MAISGET	V	Retrieves details of the specified session into user variables
&MAISPUT	V	Updates MAI session list entries
&MASKCHK	B	Tests a data string against a wildcard mask
&MSGCONT	V	Resumes normal processing of a message delivered to MSGPROC
&MSGDEL	V	Deletes a message being processed by MSGPROC

Table A-1. Verb and Built-in Function Summary Table (Sheet 4 of 6)

Name	Verb or Built-in	Description
&MSGREAD	V	Makes the next message available to MSGPROC
&MSGREPL	V	Replaces the text of a message delivered to MSGPROC
&NBLSTR	B	Removes leading and trailing blanks from a string
&NDBADD	V	Adds a record to a NetMaster Database
&NDBCLOSE	V	Signs-off (disconnects) from a NetMaster Database
&NDBCTL	V	Alters NCL/EF processing characteristics
&NDBDEF	V	Adds, updates, or deletes field definitions
&NDBDEL	V	Deletes a record from a NetMaster Database
&NDBFMT	V	Defines a list of fields to be retrieved by an &NDBGET
&NDBGET	V	Retrieves a record from a NetMaster Database
&NDBINFO	V	Retrieves information about a NetMaster Database
&NDBOPEN	V	Signs on (connects) to a NetMaster Database
&NDBPHON	V	Allows you to return a phonetic value for a character string, typically a name
&NDBQUOTE	B	Places quotes around data to protect special characters.
&NDBSCAN	V	Scans a NetMaster Database for all records matching a search argument
&NDBSEQ	V	Defines, deletes, or resets a sequential retrieval path for a NetMaster Database
&NDBUPD	V	Updates a record in a NetMaster Database
&NPF*x*CHK	B	Tests a user's network partitioning authority for a resource.
&NRDDEL	V	Deletes NRD messages.
&NUMEDIT	B	Edits the format of a real number or integer.
&OSCALL	V	Provides the client/server interface to invoke object oriented services
&OVERLAY	B	Replaces a section of a data string with data from another string
&PANEL	V	Displays a full-screen panel
&PANELEND	V	Gives up exclusive control of a display window
&PARSE	V	Parses tokenized strings into variables
&PAUSE	V	Suspends an NCL process
&PPI	V	Allows exchange of data between programs
&PPOALERT	V	Generates a simulated VTAM PPO message
&PPOCONT	V	Resumes normal processing of a VTAM PPO message
&PPODEL	V	Deletes a VTAM PPO message, or blocks its delivery
&PPOREAD	V	Makes the next VTAM PPO message available to PPOPROC
&PPOREPL	V	Resumes normal VTAM PPO message processing, after replacing message text

Table A-1. Verb and Built-in Function Summary Table (Sheet 5 of 6)

Name	Verb or Built-in	Description
&PROMPT	V	Writes text to a user's terminal and awaits input
&QEXITR	V	Terminates this procedure, plus all higher levels
&REMSTR	B	Splits a data string and returns the end portion
&RETCODE	V	Returns or resets the system return code
&RETSUB	V	Returns from a subroutine within a procedure
&RETURN	V	Passes variables to a higher nesting level
&RSCCHECK	B	Tests a user's access to a resource
&SECCALL	V	Communicates with the security subsystem or the installation security exit
SELECT STATEMENT	V	Specifies an SQL result table
&SELSTR	B	Splits a data string and returns the front portion
&SETBLNK	B	Explicitly sets a variable to blank
&SETLENG	B	Sets the length of a variable
&SETVARS	V	Extracts named keywords and associated data from a data string
&SMFWRITE	V	(MVS only) Writes a record to the SMF dataset
&SNAMS	V	Provides the client/server interface to invoke object oriented services
&SOCKET	V	Provides NCL control over allocation and management of communications using TCP/IP
&SQL	V	Requests database functions from an external database via the SQL interface
&SQLFMT	V	Formats long SQL statements
&STR	B	Assigns a multi-word string
&SUBSTR	B	Extracts part of a variable or constant
&TBLSTR	B	Removes trailing blanks from a string
&TRANS	B	Translates characters within a string
&TYPECHK	B	Tests variables and returns their type
&VARTABLE	V	Creates and maintains vartables and vartable entries
&WRITE	V	Writes a message
&WTO	V	Issues an MVS or VM/GCS WTO
&WTOR	V	Issues an MVS WTOR and waits for a reply
&ZAMCHECK	B	Indicates whether support is enabled in Management Services for a specified access method
&ZFEATURE	B	Returns availability status of a Network*IT* or SOLVE feature
&ZNCLKWD	B	Indicates if string is NCL keyword

Table A-1. Verb and Built-in Function Summary Table (Sheet 6 of 6)

Name	Verb or Built-in	Description
&ZOSCHK	B	Indicates whether support is enabled in Management Services for a specified operating system
&ZPSKIP	V	Sets new active panel skip data
&ZQUOTE and &ZQUOTE2	B	Places quotes around a string
&ZSHRINK	B	Removes leading and trailing spaces and reduces multiple spaces within a string
&ZSOCINFO	B	Obtains information about the specific socket owned by the process
&ZSUBST	B	Returns a string with substituted data
&ZSYSPARM	B	Returns the value of a systems parameter (SYSPARMS)
&ZTCPERDS	B	Returns a short message for a TCP/IP error code
&ZTCPERNM	B	Returns the logical name of a TCP/IP error code
&ZTCPINFO	B	Obtains information about the local host or TCP/IP vendor stack
&ZTCPSUPP	B	Determines if a function is supported by the current TCP/IP vendor stack
&ZUNQUOTE	B	Removes one level of quotes from a string and undoes &ZQUOTE

B

NCL System Variable List

This appendix provides a quick reference showing the *system variables* available within Management Services for use with the NCL language. For a full description of each system variable and its use, see the *Network Control Language Reference* manual.

The system variables listed in this appendix are general purpose variables. There are also special system variables that occur only on completion of certain verbs, and that provide message profile information. All of these special variables are documented in Appendix D, *System Level Procedures: Message Profiles*.

Summary Table

Table B-1. System Variable Summary Table (Sheet 1 of 12)

Name	Description
&ALLPARMS	A user variable that supplies a single string for all parameters specified when an NCL procedure is invoked
&AOMACCT1-4	Four system variables which return, for some MVS-sourced messages, the first four MVS accounting fields from the JOB statement
&AOMALARM	Returns the alarm attribute for the current message
&AOMASID	Returns the MVS address space ID (ASID) that issued the current message
&AOMATEXT	Returns the text of the current line of a message that has been delivered to AOMPROC
&AOMAUTH	Indicates whether the issuer of a WTO/WTOR is authorized
&AOMAUTO	Returns the value of the automation flag
&AOMAUTOT	Returns the value of the automation token
&AOMBC	Indicates whether or not the current message is a broadcast message
&AOMCHAR1	Returns the MVS screen character that indicates the status of operator console format messages
&AOMCOLOR	Indicates the color attribute of the current message
&AOMCONNM	Returns the Extended MCS console name
&AOMDESC	Returns the descriptor code(s) assigned to the current message, in list format
&AOMDHEX	Returns the descriptor code(s) assigned to the current message, in hexadecimal
&AOMDMASK	Returns the descriptor code(s) assigned to the current message, in &MASKCHK format
&AOMDOM	Indicates whether or not the current message is a Delete Operator Message notification (DOM-Notify)
&AOMDOMID	Returns the MVS assigned Delete Operator Message (DOM) ID of the current message
&AOMEVCLS	Returns the EVENT class value
&AOMHLITE	Returns the highlight attribute for the current message
&AOMID	Returns an ID that has been assigned by the screening table to the current message or event
&AOMIJOBN	Returns the MVS job name of the address space that issued the WTO, WTOR, or EVENT
&AOMINTEN	Returns the intensity attribute for the current message
&AOMJOBCL	Returns the MVS job class of the job that issued the WTO or WTOR
&AOMJOBID	Returns the JES job number that issued the current message
&AOMJOBNM	Returns the job name of the active address space that issued the current message
&AOMJSTCB	Returns the hexadecimal address of the job step TCB that either issued the current WTO or WTOR, or owns the TCB that issued the message

Table B-1. System Variable Summary Table (Sheet 2 of 12)

Name	Description
&AOMLDID	Returns the domain ID of the last handler of this message, event, or DOM-Notify
&AOMLROUT	Returns the local routing option for the message or event as set by the screening table ROUTE or LCLROUTE operands
&AOMLRSLT	Returns the eight LOOKUP results from screening, in &MASKCHK format
&AOMLRSL1-8	Eight system variables that return the results of up to eight LOOKUP statements
&AOMLTCTL	Indicates whether or not the current line of the current message is a *control* line
&AOMLTDAT	Indicates whether or not the current line of the current message is a *data* line
&AOMLTEND	Indicates whether or not the current message is an *end* line
&AOMLTLAB	Indicates whether or not the current line of the current message is a *label* line
&AOMMAJOR	Indicates whether or not the current line of the current message is a *major* line
&AOMMHEX	Returns the MCS flag(s) assigned to the current WTO or WTOR
&AOMMINOR	Indicates whether or not the current line of the current message is a *minor* line
&AOMMMASK	Returns the MCS flags assigned to the current message in &MASKCHK format
&AOMMONIT	Indicates whether or not the current message is also to be delivered to monitor class message receivers
&AOMMPFSP	Indicates whether or not the current message was initially suppressed by the MVS Message Processing Facility (MPF)
&AOMMSGCD	Returns the message code assigned to this message
&AOMMSGID	Returns the extracted message ID of the current message
&AOMMSGLV	Returns the highest message level of the current message
&AOMMVCON	Returns the ID of the MVS console to which the current message was routed
&AOMMVSDL	Indicates whether or not the screening table has deleted the current message from MVS
&AOMNMCON	Returns the console ID to which the current message was routed
&AOMNMDOM	Returns the assigned DOMID associated with a DOM-notify message
&AOMNMIN	Returns the number of minor lines in a multi-line WTO
&AOMNRD	Indicates whether or not the current message is to be displayed as a non-roll-delete message on OCS consoles
&AOMODID	Returns the domain ID of the system where the message originated, as set by the NMDID JCL parameter
&AOMRCLAS	Returns the ISR remote classes, as set by the screening table, in MASKCHK format
&AOMRCLS1-8	Eight system variables which provide the individual values of the eight AOM ISR remote classes for this message or event
&AOMREISS	Returns the value YES if the current message was reissued on a JES3 GLOBAL processor, otherwise its value is NO
&AOMRHEX	Returns the routing code(s) assigned to the current message, in hexadecimal

Table B-1. System Variable Summary Table (Sheet 3 of 12)

Name	Description
&AOMRKEY	Returns the retrieval key attribute
&AOMRMASK	Returns the routing code(s) assigned to the current message, in &MASKCHK format
&AOMROUTC	Returns the routing code(s) assigned to the current message
&AOMROUTE	Returns the routing option for the current message, as set by the screening table
&AOMRROUT	Returns the remote routing option for the current message, as set by the screening table
&AOMRWTOR	Indicates whether or not the current message is a Replied-to-WTOR
&AOMSALRT	Indicates whether or not the current message was sourced by the &AOMALERT verb
&AOMSDATA	Returns the saved data from a successful LOOKUP statement
&AOMSINGL	Indicates whether or not the current message is a single *line* message
&AOMSOLIC	Indicates whether or not the current message is a solicited message
&AOMSOLTP	Returns the solicit type of the current message
&AOMSOS	Identifies the type of operating system that sourced this message
&AOMSUBTP	Returns the subtype of the current line of the current message
&AOMTEXT	Returns the major text of the current message
&AOMTIME	Returns the timestamp of the current message
&AOMTYPE	Identifies the current message as a WTO, WTOR, DOM, or EVENT
&AOMUFLGS	Returns the eight user flags in &MASKCHK format
&AOMUFLG1-8	Eight system variables which are user-defined flags, set by the screening table
&AOMVMMCL	Returns the VM IUCV message class of a VM-sourced message
&AOMVMSRC	Returns the AOM/VM message source
&AOMVMUID	Returns the VM user ID that a message originated from
&AOMVMUND	Returns the VM RSCS node that a message originated from
&AOMWRID	Returns the WTOR reply ID of the current message
&AOMWRLEN	Returns the length of the text that can be passed in reply to a WTOR
&AOMWTO	Indicates whether or not the current message is a Write to Operator (WTO)
&AOMWTOR	Indicates whether or not the current message is a WTOR
&BROLINE*n*	A series of system variables that return the current broadcast lines
&CURSCOL &CURSROW	System variables that return the cursor location
&DATE*n*	A series of system variables that return the current system date in different formats
&DAY	Returns the current day of the week
&FILEID	Returns the name of the file currently being processed

Table B-1. System Variable Summary Table (Sheet 4 of 12)

Name	Description
&FILEKEY	Indicates an NCL process's current position within a UDB
&FILERC	Indicates the success or otherwise of a file processing function
&FILERCNT	Provides a count of the number of records deleted by &FILE DEL processing
&FSM	Indicates if the issuing procedure has access to a real window
&INKEY	Returns a value representing the key last used to enter data
&LOOPCTL	Returns the current setting of the automatic loop control counter
&LUCOLS	Indicates the number of columns currently allocated to this processing window
&LUEXTCO	Indicates whether the terminal supports extended color
&LUEXTHI	Indicates whether the terminal supports extended highlighting
&LUNAME	Returns the name of the terminal at which the NCL procedure is executing
&LUROWS	Returns the number of rows currently allocated to this process window
&MAI#SESS	Returns the number of currently defined sessions (equivalent to &MAINSESS)
&MAIAE	Indicates the availability of the A and E primary commands
&MAIAPPL	Returns the name of the application acting as the PLU on the MAI session
&MAICOLS	Returns the number of columns in the current MAI session's screen
&MAICROWS	Returns the number of rows in the current MAI session's screen
&MAIDISC	Indicates whether MAI will honor a terminal disconnect request
&MAIFRLU	Returns the direction of the last datastream
&MAIIKEY	Indicates the value of the key used to enter data
&MAILOCK	Indicates whether MAI will honor a terminal lock request
&MAILU	Returns the name of the VTAM APPL being used as the secondary LU
&MAIMNFMT	Returns the current menu format as long or short
&MAINSESS	Returns the number of currently defined sessions (equivalent to &MAI#SESS)
&MAIOCMD	Returns the outbound datastream sent by the PLU
&MAIREQ	Returns the MAI logon request
&MAISCANL	Returns the scan limit for session commands
&MAISID	Returns the session ID of the session of whose behalf the script is running
&MAISKIPP	Returns the system-wide value for the session command prefix character
&MAISKPK1	Returns the session command function key 1
&MAISKPK2	Returns the session command function key 2
&MAISMODE	Returns the mode in which the script procedure is running
&MAITITLE	Returns the title that is displayed at the top of the MAI-FS main menu
&MAIUNLCK	Indicates whether the datastream just received will unlock the keyboard
&MAIWNDOW	Indicates the MAI-FS session's visibility

Table B-1. System Variable Summary Table (Sheet 5 of 12)

Name	Description
&NDBERRI	Returns additional information about an NDB warning or error condition
&NDBRC	Indicates the success or otherwise of an &NDB*xxx* NCL statement
&NDBRID	Returns the record ID of the current or new record
&NDBSQPOS	Returns the relative position in an &NDBSCAN-built sequence
&NEWSAUTH	Indicates whether a user is authorized for NEWS functions
&NEWSRSET	Indicates whether the user is authorized for NEWS reset (delete) functions
&NMID	Returns the ID of this system
&OCSID &OCSIDO	Indicates the OCS ID name for the current window
&PANELID	Indicates the name of the current panel
&PARMCNT	Returns the count of the number of variables entered when a procedure was invoked
&RETCODE	Returns the current system return code
&SYSID	Returns the current operating system identification
&TIME	Returns the current time
&USERAUTH	Returns the command authority of the user who initiated the procedure
&USERID	Returns the user ID of the user currently executing the procedure
&USERPW	Returns the PASSWORD of the user
&VSAMFDBK	Returns the VSAM return code from a file processing operation
&ZACBNAME	Returns the primary VTAM ACB name in use by the system
&ZAMTYPE	Returns the name of the access method used to connect the terminal on which the NCL procedure is executing
&ZAPPCACC	Returns the number of active APPC conversations for the NCL process
&ZAPPCCSI	Returns the Client/Server indicator for the APPC conversation
&ZAPPCELM	Returns the message from an Error Log GDS variable received after an error, or deallocate abend
&ZAPPCELP	Returns product set information from an Error Log GDS variable received after an error, or deallocate abend
&ZAPPCID	Returns the conversation ID which identifies an APPC conversation (a unique integer)
&ZAPPCIDA	Returns the APPC conversation ID for the transaction that started the NCL process
&ZAPPCLNK	Returns the link name for an APPC conversation
&ZAPPCMOD	Returns the modename for an APPC conversation
&ZAPPCPCC	Returns the number of pending APPC conversations for the NCL process
&ZAPPCQLN	Returns the network qualified *local* LU name

Table B-1. System Variable Summary Table (Sheet 6 of 12)

Name	Description
&ZAPPCQRN	Returns the network qualified *remote* LU name
&ZAPPCRM	Returns the current receive map name
&ZAPPCRTS	Indicates whether or not a request to send has been received
&ZAPPCSCM	Returns the Server Connection Mode indicator
&ZAPPCSM	Returns the current send map name
&ZAPPCSND	Returns the APPC SEND protocol indicator
&ZAPPCSTA	Returns the current state of an APPC conversation
&ZAPPCSYN	Returns a character string, equivalent to that of the SYNC_LEVEL parameter of the LU6.2 MC_GET_ATTRIBUTES verb
&ZAPPCTRN	Returns the locally known transaction identifier (up to 32 characters) for an APPC conversation
&ZAPPCTYP	Returns a character string providing the APPC conversation type
&ZAPPCVRB	Returns the last APPC verb that was issued
&ZAPPCWR	Returns a character string, equivalent to that of the LU6.2 WHAT_RECEIVED parameter
&ZAPPCWRI	Returns a character string, equivalent to that of the LU6.2 WHAT_RECEIVED parameter
&ZBLANK1	Returns a single blank character
&ZBROID	Returns the broadcast identifier associated with the NCL process
&ZBROTYPE	Indicates the type of broadcast associated with the issuing procedure
&ZCOLS	Indicates the number of columns associated with the physical terminal
&ZCONSOLE	Returns the MVS console number associated with a console user ID
&ZCURSFLD &ZCURSPOS	Returns the name of the field where the cursor is positioned and the offset within that field
&ZDBCS	Indicates whether a terminal supports double byte character set datastreams (DBCS)
&ZDB2REL	Returns the DB2 version number
&ZDOMID	Returns the deletion identifier for a non-roll delete message
&ZEDBRC	Returns a return code from the external database
&ZEDBREAS	Returns a reason code from the external database
&ZFDBK	Returns completion information following execution of selected NCL statements
&ZDSNQLCL	Returns the value of the local dataset qualifier
&ZDSNQSHR	Returns the value of the shared dataset qualifier
&ZGDATE*n*	A set of system variables that return the date, in different formats, based on GMT
&ZGDAY	Returns the day of the week, based on GMT
&ZGOPS	Indicates the generic type of operating system

Table B-1. System Variable Summary Table (Sheet 7 of 12)

Name	Description
&ZGTIME*n*	A set of system variables that return the time, based on GMT
&ZGTIMEZ*n*	A set of system variables that indicate the difference in time between local (operating system) time and GMT
&ZINTYPE	(Message profile variable) Specifies whether an &INTREAD operation has been satisfied by a request message or a response message
&ZINVOKE	Returns the &OSCALL invocation identifier
&ZIREQCNT	Returns the count of messages queued to an NCL process's *dependent request queue*
&ZIRSPCNT	Returns the count of messages queued to an NCL process's *dependent response queue*
&ZJOBNAME	Returns the job name
&ZJOBNUM	Returns the JES2/3 job number for the last job submitted by NCL (OS/VS only)
&ZJRNLACT	Returns the DD name of the active journal dataset
&ZJRNLALT	Returns the DD name of the alternate (or inactive) journal dataset
&ZLCLIPA	Returns the IP address of the local host for a TN3270 session
&ZLCLIPP	Returns the IP port of the TN3270 server for a TN3270 session
&ZLOGMODE	Returns the name of the VTAM *logmode table entry* used when the current terminal was connected
&ZLUNETID	Returns the network ID of the currently connected terminal
&ZLUTYPE	Indicates the type of device or environment
&ZLU1CHN	Indicates the segment position of a message received from an LU1 device
&ZMAIACT# or &ZMAIACTN	Returns the number of active sessions associated with the current window
&ZMALARM	Indicates whether the message will cause the terminal alarm to sound
&ZMALLMSG	Indicates whether the message was generated by a MSG ALL command
&ZMAOMAU	Indicates whether or not the original WTO or WTOR issuer was authorized
&ZMAOMBC	Indicates whether or not the current message has the AOM broadcast attribute
&ZMAOMDTA	Indicates whether or not the current message contains AOM data
&ZMAOMID	Returns the AOM ID value
&ZMAOMJI	Returns the MVS JOBID of AOM messages sourced from MVS
&ZMAOMJN	Returns the MVS job name of AOM messages sourced from MVS
&ZMAOMMID	Returns the AOM message ID
&ZMAOMMIN	Indicates whether or not this is an AOM minor line
&ZMAOMMLC	Indicates whether or not the current message is an MLWTO control line
&ZMAOMMLD	Indicates whether or not the current message is an MLWTO data line
&ZMAOMMLE	Indicates whether or not the current message is an MLWTO end line

Table B-1. System Variable Summary Table (Sheet 8 of 12)

Name	Description
&ZMAOMMLL	Indicates whether or not the current message is an MLWTO label line
&ZMAOMMLT	Indicates the type of MLWTO of the current message
&ZMAOMMLV	Returns the highest AOM message level of the current message
&ZMAOMMSG	Indicates whether or not the current message was marked for propagation to eligible AOM receivers
&ZMAOMRC	Returns the AOM routing code(s) assigned to the current message
&ZMAOMRCM	Returns the routing code(s) assigned to the current message, in &MASKCHK format
&ZMAOMRCX	Returns the AOM routing code(s) assigned to the current message, in hexadecimal characters
&ZMAOMSOS	Returns the operating system type from which the current AOM message came
&ZMAOMSYN	Returns the originating system name for the current message
&ZMAOMTM	Returns the AOM timestamp of the current message
&ZMAOMTYP	Returns the AOM type of a message
&ZMAOMUFM	Returns the eight AOM user flags in &MASKCHK format
&ZMAOMUF1-8	Eight system variables which return the AOM user defined flags, set in the screening table
&ZMAOMUI	Returns the originating user ID of an AOM message from a VM system
&ZMAOMUN	Returns the VM RSCS node name that an AOM/VM message came from
&ZMAPNAME	(Message profile variable) Returns the mapname for the embedded user MDO in the current $MSG MDO if present
&ZMCOLOR &ZMCOLOUR	Returns the color attribute of the message
&ZMDOCOMP	Returns the last name segment of the fully qualified name for the MDO component involved in the last operation
&ZMDOFDBK	Returns the feedback code after any verb references an MDO
&ZMDOID	Returns the identifier of the MDO involved in the last operation
&ZMDOM	Indicates whether the message is a *delete operator message* instruction
&ZMDOMAP	Returns the map name for &ZMDOID
&ZMDOMID	Returns the *delete operator message identifier* (domid) of the message read, provided the message has the non-roll delete message attribute (as determined by the setting of the &ZMNRD terminal)
&ZMDONAME	Returns the fully qualified name of the MDO component involved in the last operation
&ZMDORC	Returns the return code after any verb references an MDO (used in conjunction with &ZMDOFDBK)
&ZMDOTAG	Returns the MDO tag value of the component involved in the last operation
&ZMDOTYPE	Returns the ASN.1 type of &ZMDOCOMP

Table B-1. System Variable Summary Table (Sheet 9 of 12)

Name	Description
&ZMEVONID	Returns the *nclid* of the procedure which issued the &EVENT which caused the message on the RESP queue
&ZMEVPROF	Returns the EDS profile name which resulted in delivery of an event notification.
&ZMEVRCDE	Returns the *routecode* of an incoming event message
&ZMEVTIME	Returns the time that an event originated, in the format HH.MM.SS.THT
&ZMEVUSER	Returns the user ID of a user who issued the &EVENT verb which caused the message on the RESP queue
&ZMHLIGHT &ZMHLITE	Returns the display highlighting attribute of the message. Values are NONE, USCORE, REVERSE, or BLINK
&ZMINTENS	Returns the display intensity attribute of the message. Values are HIGH, or LOW, or null if no message is processed
&ZMLNODE	Returns the terminal name of the user to whom the log message is to be attributed
&ZMLOGCMD	Returns whether a log message is an echo to the log of a command (available to &LOGREAD only)
&ZMLSRCID	Returns the message prefix of the last handler for the message just received
&ZMLSRCTP	Returns the type of the last handler for the message just received
&ZMLTIME	Returns the time stamp of a log message (available to &LOGREAD only) (format HH.MM.SS.THT)
&ZMLUSER	Returns the user ID the log message came from (available to &LOGREAD only)
&ZMMONMSG	Indicates whether the message received is a monitor class message
&ZMMSG	Indicates whether or not the message received is a standard message
&ZMMSGCD	Indicates the hexadecimal message code attribute for this message
&ZMNMDIDL	Returns the domain ID for the previous system to handle this message
&ZMNMDIDO	The domain ID for the system where this message originated
&ZMNRD	Indicates whether the message carries the non-roll delete attribute
&ZMNRDRET	Indicates whether the message has been received as a result of a NRDRET command being issued by the user
&ZMODFLD	Returns the name of the next modified field on a panel
&ZMOSRCID	Returns the message prefix for the originator of the message just received
&ZMOSRCTP	Returns the type for the originator of the message just received
&ZMPPODTA	Indicates whether any PPO message profile information is available regarding this message
&ZMPPOMSG	Indicates whether the message originated from PPO
&ZPPOSCNT	A counter of remote domains to which a PPO message was delivered.
&ZMPPOSEV	If &ZMPPODTA=YES, then this variable includes gives the severity level of the PPO message
&ZMPPOTM	If &ZMPPODTA=YES, this variable gives the time when the message was created

Table B-1. System Variable Summary Table (Sheet 10 of 12)

Name	Description
&ZMPPOVNO	If &ZMPPODTA=YES, this variable returns the VTAM message number for the PPO message
&ZMPREFXD	Indicates whether the message text has been prefixed with identifier values
&ZMPTEXT	Returns the message text, prefixed according to the current profile settings
&ZMREQID	Returns either user ID or NCLID, if &ZINTYPE=REQ
&ZMREQSRC	Returns the source of the INTQ command if &ZINTYPE=REQ
&ZMSOLIC	Indicates whether the message was solicited or unsolicited
&ZMSOURCE	Returns the verb that last set the values for the message profile variables
&ZMTEXT	Returns the text of the message received
&ZMTYPE	Returns the type of message received
&ZNCLID	Returns the unique identifier of the NCL process
&ZNCLNEST	Returns the current procedure's EXEC nesting level within method level
&ZNCLTYPE	Returns the type of the current procedure
&ZNCLV	Returns the current release level of NCL
&ZNETID	Returns the value of the VTAM network identifier
&ZNETNAME	Returns the network name of the primary ACB
&ZNMDID	Returns the value of the *domain identifier*
&ZNMSUP	Returns the value of the *system user prefix*
&ZOCS	Indicates whether the NCL process is associated with an OCS window
&ZOPS	Returns the type of operating system
&ZOPSVERS	Returns the version of the operating system
&ZOSREQID	Returns a unique request identifier associated with an object services conversation
&ZOUSERID	Returns the originating user ID for an NCL process
&ZPERRORC	Returns the value of the standard panel field attribute 'COLOR' for error fields and messages
&ZPERRORH	Returns the value of the standard panel field attribute 'HLITE' for error fields
&ZPINPHIC	Returns the value of the standard panel field attribute 'COLOR' for mandatory input data fields and command fields
&ZPINPLOC	Returns the value of the standard panel field attribute 'COLOR' for optional input data fields
&ZPINPUTH	Returns the value of the standard panel field attribute 'HLITE' for data input fields
&ZPINPUTP	Returns the value of the standard panel field attribute 'PAD' for data input fields
&ZPLABELC	Returns the value of the standard panel field attribute 'COLOR' for field labels and comments
&ZPMTEXT1	Returns the text of the Primary Menu broadcast

Table B-1. System Variable Summary Table (Sheet 11 of 12)

Name	Description
&ZPOUTHIC	Returns the value of the standard panel field attribute ‘COLOR’ for output data fields that are always present
&ZPOUTLOC	Returns the value of the standard panel field attribute ‘COLOR’ for output data fields that are not always present
&ZPPFKEYC	Returns the value of the standard panel field attribute ‘COLOR’ for the output fields on the left and right of the panel title and the function key area
&ZPPI	Indicates whether or not PPI is available
&ZPPINAME	Returns the defined receiver-ID of the current NCL process, if it has one
&ZPRODNAM	Returns the product name
&ZPSERVIC	Returns the value of the first four bytes of the PSERVIC field of the BIND for the current terminal
&ZPSKIP	Returns the next available segment of panel skip data
&ZPSKPSTR	Returns the current panel skip string in its entirety
&ZPSUBTLC	Returns the value of the standard panel field attribute ‘COLOR’ for sub-titles, headings and trailers
&ZPTITLEC	Returns the value of the standard panel field attribute ‘COLOR’ for the panel title
&ZPTITLEP	Returns the value of the standard panel field attribute ‘PAD’ for the panel title
&ZPWSTATE	Returns the state of a user’s password
&ZREMIPA	Returns the IP address of the remote host for a TN3270 session
&ZREMIPP	Returns the remote host IP port for a TN3270 session
&ZROWS	Returns the number of rows available to the physical terminal
&ZSCOPE	Returns the scope of the server name if the current NCL process is registered as a server
&ZSECEXIT	Returns the type of security exit installed
&ZSERVER	Returns the server name if the current NCL process is registered as a server
&ZSNAMID	Returns an integer when using the &SNAMS verbs
&ZSOCCID	Returns the socket ID used by the interface
&ZSOCERRN	Returns the error number value associated with the last referenced socket
&ZSOCFHNM	Returns the full host name of the host referenced by some requests
&ZSOCHADR	Returns the IP address of the host referenced by some requests
&ZSOCHNM	Returns the host name of the host referenced by some requests
&ZSOCID	Returns the socket ID of the last referenced socket
&ZSOCPRT	Returns the port number of the last referenced socket
&ZSOCTYPE	Returns the type of the last referenced socket
&ZSOCVERR	Returns vendor error information from the last referenced socket
&ZSQLCODE	Returns an SQL code after the execution of an &SQL verb

Table B-1. System Variable Summary Table (Sheet 12 of 12)

Name	Description
&ZSQLSTAT	Returns the SQL state, which is set after the execution of the &SQL verb
&ZSSCPNAM	Returns the value of the VTAM SSCP name
&ZSYSNAME	Returns the MVS SYSNAME value
&ZTCP	Returns the status of the TCP/IP API
&ZTCPHSTA	Returns the value of the local host's IP address
&ZTCPHSTF	Returns the value of the local host's full name
&ZTCPHSTN	Returns the value of the local host's short name
&ZTIME*n*	Returns system variables for different formats of the current time
&ZTSOUSER	Indicates if the user has connected through the TSO or TSS interface
&ZUCENAME	Returns the UCE name which the SOLVE region is using to communicate with XNF
&ZUDATE*n*	A set of system variables that return the user's date, in different formats, time zone adjusted
&ZUDAY	Returns the user's day of the week, time zone adjusted
&ZUSERLC	Returns the language code for this user
&ZUSERSLC	Returns the system recognized language code for a user
&ZUSRMODE	Returns a value indicating special conditions of this signed on user
&ZUTIME*n*	A set of system variables that return the user's time, time zone adjusted
&ZUTIMEZ*n*	A set of system variables that indicate the difference in time between local (operating system) time and the user's time zone
&ZUTIMEZN	Returns the user's time zone name
&ZVARCNT	Returns the number of variables created or modified by the last NCL verb that used generic processing
&ZVTAMLVL	Returns the VTAM release and version number, if available
&ZVTAMPU	Returns the host PUname of VTAM
&ZVTAMSA	Returns the subarea number of VTAM
&ZWINDOW	Returns the identifier of the current window
&ZWINDOW#	Returns the number of active windows
&ZWSTATE	Returns a value indicating the current window's state
&0	Returns the name of the procedure currently being executed
&00	Returns the name of the base procedure of the NCL process
&000	Returns the system global variable prefix

C

NCL VSAM Techniques

NCL is designed to use VSAM facilities efficiently. It is structured to support many users concurrently and uses techniques such as dynamic generation of VSAM control blocks, automatic load processing, automatic verify and optional use of LSR pools.

This appendix describes the techniques used by NCL. Some knowledge of VSAM is assumed.

Initialization and ACB Open Processing

In OS/VS systems, at initialization the system scans for DD names that commence with the 3-character UDB, to identify UDBs that are to be automatically opened. Using the 3-character UDB prefix is not mandatory for UDBs. It acts only as a means of identifying those UDBs which are to be automatically opened during initialization.

Additional UDBs can be opened at any time using the UDBCTL command. When using the UDBCTL command, an open can be performed for any UDB, regardless of the DDNAME. In VSE/SP systems, UDBs must be opened using the UDBCTL command.

With certain UDBs, you might want to use some of the more advanced VSAM facilities that Management Services supports. For example, you might want to specify use of the LSR pool or sequential insert strategy (SIS) to improve system performance.

Use of these optional facilities must be requested on the UDBCTL command at open time. Therefore, if the UDB is opened automatically during system initialization, selection of these options is not possible.

In OS/VS systems, where a mix of both automatically opened UDBs and those requiring special options is required, you need only to choose a DD name that does not commence with the three character UDB prefix (thus bypassing automatic open processing) and then place a UDBCTL command in either the NMINIT or NMREADY procedures, specifying the required options. For example:

```
UDBCTL OPEN=MYFILE ID=MYFILEID SIS LSR
```

Note

In a VSE environment, no automatic open processing is performed during system initialization. UDBs must be opened using the UDBCTL command. These commands are normally placed in the NMINIT or NMREADY procedures that are executed during the latter stages of system startup.

The UDBDEFER JCL parameter can be specified if automatic open processing is to be bypassed entirely in OS/VS systems. In this case all open processing for UDBs must be initiated by use of the UDBCTL command.

During the open process, a VSAM ACB is generated for each dataset and an open attempted. If the open fails, the system internally links to IDCAMS to perform a verify of the dataset. Therefore, it is not necessary to include verify steps in the system JCL.

Automatic Verification and Loading

On completing the verify function, the open is re-attempted. If the dataset is empty, the system closes and re-opens the dataset in create mode and attempts to perform a load according to the following rules:

For Entry Sequence Datasets (ESDS)

For an ESDS a single record in the following format is loaded:

```
N28510 VSAM INITIAL LOAD PERFORMED AT hh.mm.ss
ON day-dd.mon.year
```

This initial load record remains in the ESDS and if necessary can be skipped during off-line processing by using the IDCAMS utility to REPRO the dataset specifying the SKIP=1 operand.

Note

This load process is performed only if the dataset is classified by VSAM as being empty. This implies that the high-order RBA is 0. If data exists within the dataset when it is opened, then the load process is bypassed.

In an MVS environment, where output is sent directly to JES2 or JES3, no load processing is required and hence the N28510 message is not the first record in the file.

If the load fails, the dataset is classified as unusable and is blocked from further processing until the problem is corrected. The SHOW UDB command can be used to determine the current status of a UDB and displays any error code detected during open processing. If necessary, the dataset in an MVS environment can be de-allocated using the DEALLOC command to assist in off-line correction of problems.

For Key Sequenced Datasets (KSDS)

For a KSDS, a single record with a key of all X‘00’ is inserted into the dataset. If the load is successful, the dataset is closed and re-opened in *update* mode and the initial load record deleted. If the load fails, the dataset is classified as unusable and is blocked from further processing. The SHOW UDB command can be used at any time to determine the status of a UDB and if necessary to obtain any error code set during open processing.

Note

The loading of alternate indices by Management Services is not supported. This must be performed using the IDCAMS BLDINDEX function. Once built, these indices are maintained correctly.

RPL Handling

Until a file is opened (using an &FILE OPEN statement) by an NCL procedure, no RPLs are created and no work buffers allocated.

Before a UDB can be referenced by a procedure, the UDBCTL command must be used to assign a logical file ID to the physical dataset. This command can be included in the NMINIT procedure or entered from OCS.

To open a file, an NCL procedure uses an &FILE OPEN statement. This statement references the logical file ID previously assigned by the UDBCTL command, and in doing so completes the link between the NCL procedure and the actual dataset.

It is at this time that a VSAM RPL is created for use by the NCL procedure. This RPL is used in all subsequent requests for this dataset by the procedure. The RPL remains in existence until either explicitly freed by the &FILE CLOSE statement, or termination of the NCL procedure. When processing with multiple files, one RPL is created for each new file ID the first time that the file is opened.

Obtaining I/O Buffers

At the time the RPL is generated an I/O buffer is also obtained. This buffer is large enough to hold the largest record that could be read from, or written to, the UDB. Therefore, it is important to accurately define the record sizes when the VSAM cluster is allocated. Specification of record sizes in excess of that required not only impacts VSAM algorithms for space allocation, but also forces Management Services to obtain buffers of an unnecessary size, thus wasting storage.

Concurrent Access to Multiple UDBs

An NCL procedure can actively process several UDBs at a time. The mandatory ID= operand on the &FILE verbs, indicates which UDB the verb is to be actioned upon. Each file must be opened separately, using an &FILE OPEN statement before reads and writes can be performed on it. There is no overhead in processing several files simultaneously, because NCL merely swaps pointers between the work buffers and RPLs used for each file, according to the ID specified on the &FILE verb.

Current keys and dataset positioning are remembered by NCL when processing swaps from one file to another, so it is not necessary for the NCL procedure to remember these.

The example in the next section, of copying one file to another, shows how processing can be alternated between two files.

Dataset Positioning and Generic Retrieval

NCL supports both sequential and generic retrieval from keyed datasets. Such functions imply that a current position within the file is maintained. Thus, the NCL procedure can simply request the next record and it is supplied.

For example:

```
&FILE OPEN ID=FILE1 FORMAT=DELIMITED   -* Open file 1
&FILE OPEN ID=FILE2 FORMAT=DELIMITED   -* Open file 2
.LOOP
   &FILE GET ID=FILE1 OPT=SEQ VARS=A* RANGE=(1,6)
                                       -* Read record
   &IF &FILERC NE 0 &THEN &GOTO .ENDPROC
                                       -* End if not 0
   &X = &FILEKEY                       -* Copy key of
                                       -* record that was
                                       -* read into
                                       -* variable
   &FILE PUT ID=FILE2 KEYVAR=X VARS=(A1,A2,A3,A4,A5,A6)
                                       -* Write record

   &GOTO .LOOP                         -* Loop to read
                                       -* next record on
                                       -* file 1
```

Note

This example does not fully cater for such things as error conditions.

It is not necessary for the NCL procedure to increment keys.

Under certain circumstances, such as with generic retrieval, it might be necessary to alter the retrieval sequence and commence retrieval using a different key.

NCL must be informed that such a change is required and that the current retrieval sequence is to be stopped. This is done using the &FILE GET ID=*fileid* OPT=END statement. This indicates to NCL that generic retrieval is to be terminated in anticipation of some other processing.

If an end-of-file condition is signalled, no &FILE GET ID=*fileid* OPT=END is required. The use of a non-generic function, such as the specific retrieval of a record, also cancels a previous generic function.

Releasing File Processing Resources

The &FILE OPEN statement allocates certain resources to the requesting NCL procedure. It is not normally necessary to release file processing resources within an NCL procedure. This is performed automatically when the NCL procedure terminates.

Under certain circumstances, such as in an EASINET procedure, where there can be many concurrent users performing file processing, it might be desirable to release any file processing overheads when they are no longer required, to ensure that system overheads are minimized.

This can be done with the &FILE CLOSE statement. &FILE CLOSE allows either specific files or all files to be freed. When this is done, any storage associated with processing the file is released and the connection is logically severed for that user.

Having used &FILE CLOSE to release a particular file, connection can be re-established using another &FILE OPEN statement.

&FILE CLOSE destroys any generic retrieval position a user might have established within a file and any subsequent reference would have to re-establish that position if required.

Displaying File Information

The SHOW UDB command can be used to display details about files available to the system. This information includes details about the number of active users, space utilization and the current status. In addition, any open error codes that caused a file to be disabled are displayed.

The SHOW VSAM command can be used to obtain additional details about the system's VSAM files. This includes record and Control Interval sizes, statistics on the number of CI and CA splits and details of any buffer or string shortages that have been experienced. In OS/VS systems, this display also provides details on the performance of the Local Shared Resource (LSR) pool if one is in use.

For a complete description of the SHOW UDB and SHOW VSAM commands and their use, see the *Management Services Command Reference* manual.

Controlling UDB Performance

The techniques used by NCL should ensure efficient processing of VSAM files. Additional performance gains can be obtained by the allocation of additional buffers and processing strings.

This is achieved using the JCL AMP statement subparameters on the DD statement for the file, or by specification of these options on the UDBCTL command:

BUFNI	The number of index buffers to be allocated by VSAM
BUFND	The number of data buffers to be allocated by VSAM
STRNO	The maximum number of concurrent strings to be used by VSAM

Additional facilities are offered as options on the UDBCTL command. They include:

SIS	Use Sequential Insert Strategy
DEFER	Use deferred writes
LSR	Use Local Shared Resource pool

Caution

These parameters should only be changed if the impact on VSAM processing is clearly understood. Inadvertent changes can impose severe storage overheads which could impact the operation of other system components. See your *VSAM Programmer's Guide* for additional details.

Off-line Processing of Datasets

In an MVS environment, the DEALLOCATE command can be used to release a UDB for off-line processing. Conversely, the ALLOCATE command can be used to bring a UDB on-line for processing.

In non-MVS environments, the stripping of files, created by NCL for further off-line processing, can be achieved without a restart of the system if the following rules are followed:

- VSAM SHAREOPTIONS must allow concurrent access to the dataset.
- DISP=SHR must be specified on the dataset.
- The UDBCTL CLOSE command is used to stop further logical connections to the file and to physically close the file. This is only successful if there are no active users currently referencing the file. The SHOW UDBUSER command can be used to display the current user of a UDB.
- Use a utility to strip the file (for example, the VSAM IDCAMS utility REPRO facility).

If the UDB has been processed using standard format, any off-line processing programs must take the high-value (X‘FF’) field separators into account when determining the format of data within records.

To allow non-UDB format files to be processed, Management Services provides a format operand on the &FILE OPEN and &FILE SET statements, that designates the format of the files being processed. See Chapter 7, *NCL File Processing*, in this manual for a detailed discussion of these facilities.

D

System Level Procedures: Message Profiles

There is a series of NCL verbs that are used to retrieve messages that are queued to particular NCL process environments. The verbs of this type that exist in Management Services are:

- &INTREAD
- &LOGREAD
- &MSGREAD
- &PPOREAD

In addition, there is the &AOMREAD verb if you have the AOM feature and &CNMREAD if you have the NEWS feature on your system.

System Level Procedure Environments

The term *system level procedure* applies to those specific NCL procedures that have access to specialized flows of information within a SOLVE system. Within this system there are several system level procedures, the principal ones being LOGPROC and PPOPROC. LOGPROC has access to and control over the flow of messages to the Management Services log. PPOPROC receives unsolicited messages from VTAM about events within the network.

The significant aspect of the special procedures is that there is only one of each in the system, and they have particular privileges and responsibilities that do not apply to the usual NCL procedures executed by Management Services users.

MSGPROC Viewed as a System Level Procedure

Each OCS window can have a MSGPROC procedure associated with it. There is only one MSGPROC per window and it has the ability to review and process every message sent to its associated OCS window from any source, before the messages are actually delivered to the window for display.

Just as LOGPROC is the only procedure in the system that can review the messages flowing to the Management Services log, so an OCS window's MSGPROC is the only NCL procedure that can review and process the message traffic flowing to that OCS window.

In this respect a MSGPROC is classified as a *system level procedure*, even though there can be many MSGPROC procedures active in a SOLVE system on behalf of many different OCS windows; MSGPROC has access to and control over a specific message flow and has privileges not open to the usual NCL procedures executed by users.

&INTREAD: The Dependent Processing Environment

The last form of privileged access to traffic flow within a SOLVE system occurs when &INTCMD and &INTREAD statements are used to execute commands or other NCL processes within an NCL *dependent processing environment*.

When a procedure executes an &INTCMD statement, command results are returned to that procedure. These results are queued and can then be read back by the original procedure through the &INTREAD statement.

&INTREAD therefore provides privileged access to the flow of command result messages produced by the various commands and NCL procedures that execute within the original procedure's dependent processing environment.

&INTREAD is also able to access unsolicited messages, such as monitor messages, for which the independent environment is profiled.

Note

NCL concepts and topics such as the dependent processing environment are described in *Chapter 2, NCL Concepts.*

Message Handling and Processing by System Level Procedures

All system level procedures that have privileged access to a particular message flow have the following capabilities:

- They can read the next message from the traffic flow by using a privileged NCL verb statement which provides access to the queue of messages. The following verbs are used by specific system level procedures:

  &LOGREAD
  Used by the LOGPROC procedure to get a copy of the next message that is to be written to the SOLVE log.

  &PPOREAD
  Used by the PPOPROC procedure to get a copy of the next message received from VTAM.

  &MSGREAD
  Used by a MSGPROC procedure to read the next message queued for display on the OCS window with which the procedure is associated.

  &INTREAD
  Used by any NCL procedure to read the next request or response queued to it via its dependent processing environment.

- They can indicate what is to be done with the message that they have received as a result of executing the 'read' statement. Typically, once a message has been received, the procedure can indicate that the message is to be processed normally (for example, &LOGCONT, &PPOCONT, &MSGCONT), or deleted and not propagated any further (for example, &LOGDEL, &MSGDEL), or replaced with an alternative message (for example, &MSGREPL).

Deciding What to Do with a Message

While the system level procedures have verbs for accessing the message flow that they are to monitor and verbs for indicating the course of action to be taken with a message when it has been read, the procedures must also contain the logic necessary to analyze the messages that are read to determine what action is required.

The logic of these procedures is usually built on the concept of filtering the message flow, searching for messages that are defined as being of interest based on arguments such as message numbers or certain values that occur within the message.

In addition to processing options based on message text content, messages can also have a large number of other attributes that are not textual in nature. For example, a message can have a color attribute that causes it to be displayed in red when it is sent to a terminal.

Other messages can have the *non-roll delete* attribute, meaning that they will not roll off an OCS window display until specifically deleted.

To assist with the logic necessary to process the text and non-text attributes of each message, when a message is read by one of the system level procedure verbs listed earlier a *message profile* is created for analysis and interrogation by the procedure.

The Message Profile

The concept of the message profile is that all the attributes of a message should be available as specific settings of a special group of system variables. The profile variables can be tested for specific values to assist the procedure in deciding whether special processing is needed for a message or whether the message is of no importance to the procedure. Since testing variables for specific values is easier than scanning text strings, examination of a message's profile provides a simpler and more efficient method of message analysis.

All profile variables are available in Mapped Data Object (MDO) format for the &INTREAD, &LOGREAD and &MSGREAD verbs. The object stem name for each verb is unique, but they are all mapped by the same map, that is $MSG.

- For &INTREAD the stem name is $INT.
- For &LOGREAD the stem name is $LOG.
- For &MSGREAD the stem name is $MSG.

The $MSG MDO can contain additional information that is not available in the profile variables.

The $MSG map definition is part of the Management Services distributed system and can be reviewed from Mapping Services (option D.M from the Primary Menu). Note that most of the data components are optional.

Message Profile Variables

Following execution of any of the &INTREAD, &LOGREAD, &PPOREAD, or &MSGREAD verbs, message profile variables are set to reflect the message profile of the message read by the verb statement. The particular variables that are set depend on the verb executed and on the class of message received.

This section describes the different values that can be set in the variables that apply to the &INTREAD, &LOGREAD, and &MSGREAD verbs, and the conditions under which each of the variables can be set. The individual variables that form the message profiles of the message flows handled by the different verbs are then cross-referenced against the individual verbs in the remaining sections of this appendix.

Note

System profile variables are available immediately after the &*xxx*READ statement is executed and are generally available until another NCL statement is executed that alters the profile. System variables are valid only within the scope of the procedure and their values are unpredictable when control is passed to another procedure level. MDOs can be shared between called procedures using SHRVARS and explicitly on the &return statement.

Following is a list of the complete set of message profile variables that apply to these three verbs. Their MDO equivalents are included in brackets, where relevant.

Note

stem can be $MSG, $INT or $LOG depending on which verb caused the variable to be set.

&ZINTYPE

Specifies whether an &INTREAD operation has been satisfied by a *request message* or a *response message*. Values are REQ (request message) or RESP (response message) or NONE (no message).

&ZMALARM (*stem*.MSGATTR.ALARM)

Indicates whether the message will cause the terminal alarm to sound. Value is YES or NO.

&ZMALLMSG

Indicates whether the message was generated by a MSG ALL command. Value is YES or NO.

&ZMCOLOUR (*stem*.MSGATTR.DISPLAY.COLOUR)

Indicates the color attribute of the message. Value can be any one of NONE, RED, BLUE, GREEN, YELLOW, TURQUOISE, PINK, or WHITE.

&ZMDOM

Indicates whether the message is a *delete operator* message instruction. Value is YES or NO. This value is dependent also on the setting of the &ZMTYPE variable.

&ZMDOMID (*stem.*DOMID)

Contains the *delete operator message identifier* (*domid*) of the message read, if the message has the non-roll delete message attribute, as determined by the setting of the &ZMNRD variable. If &ZMNRD=NO, this variable is set to null.

&ZMEVONID (*stem.*SOURCE.NCLID)

Is set when the incoming message was generated by an &EVENT verb and contains the *nclid* of the procedure which issued the &EVENT.

&ZMEVPROF (*stem.*SOURCE.EVENTPROFILE)

Is set for incoming event messages (N00102) and represents the EDS profile name which resulted in delivery of the event notification.

&ZMEVRCDE (*stem.*EVENT.ROUTECDE)

Contains the *routecde* of the incoming event message (N00102) if the ROUTECDE operand was specified on the originating &EVENT verb.

&ZMEVTIME (*stem.*SOURCE.TIME.HHMMSSTTT)

Is set for incoming event messages (N00102) to the time the event originated. Time is in the format HH.MM.SS.TTT.

&ZMEVUSER (*stem.*SOURCE.USER)

Is set when the incoming message was generated by an &EVENT verb and contains the *user ID* of the user who issued the &EVENT verb. This variable is also set for some system events and represents the user who was responsible for the event generation.

&ZMFTSMSG

Indicates whether the message originated from File Transmission Services (FTS). Value is YES or NO.

&ZMHLIGHT (*stem.*MSGATTR.DISPLAY.HLITE)

Indicates the display highlighting attributes of the message. Values are NONE, USCORE, REVERSE, or BLINK.

&ZMINTENS (*stem.*MSGATTR.DISPLAY.INTENS)

Indicates the display intensity attribute of the message. Values are HIGH or LOW or null if no message is being processed.

&ZMLNODE (*stem.*SOURCE.REGION)

Indicates the terminal name of the user to whom a log message is to be attributed. Value is the name of a terminal. (This is available to &LOGREAD only.)

&ZMLOGCMD

Indicates whether a log message is an echo to the log of a command. Value is YES or NO. (This is available to &LOGREAD only.)

&ZMLSRCID (*stem*.PREFIX.LASTMSGID.ID)

Contains the message prefix of the last handler of the message just received.

&ZMLSRCTP (*stem*.PREFIX.LASTMSGID.TYPE)

Indicates the type of the last handler of the message just received. Values can be:

null

If the message was generated within this system.

ROF

If the message was delivered across a ROF session.

MAIOC

If the message was delivered across an MAI OC session.

&ZMLTIME

Contains the timestamp of a log message. (This is available to &LOGREAD only.) Time is in the format *HH.MM.SS.TTT.*

&ZMLUSER (*stem*.SOURCE.USER)

Contains the user ID to whom generation of the log message is to be attributed. (This is available to &LOGREAD only.)

&ZMAPNAME (*stem*.MAPNAME)

Contains the mapname of the object in $MSG.USERMDO for the current message (if present).

&ZMMONMSG

Indicates whether the message received is a monitor class message. Values are YES or NO.

&ZMMSG

Indicates whether the message received is a standard message or not. The setting of this variable is always opposite to the setting of the &ZMDOM variable and is dependent on the setting of the &ZMTYPE variable.

&ZMMSGCD (*stem*.MSGATTR.MSGCODE)

Indicates the (hex) message code attribute of this message. Value can be 00-FF. The message code dictates which user IDs are eligible to receive the message.

&ZMNMDIDL (*stem*.SOURCE.LAST.DOMAIN)

The domain ID of the *last* SOLVE system to handle this message. This can be the same as the originating system, or different if the message was originated by a remote system and then routed onwards by an intermediate system.

&ZMNMDIDO (*stem*.SOURCE.ORIG.DOMAIN)

The domain ID of the SOLVE system from which this message originated. If sourced from the local system it will contain the local system's domain ID. If sourced from a remote system this variable carries the domain ID of the originating system even though the message might have been routed onwards by intermediate systems.

&ZMNRD

Indicates whether the message carries the *non-roll delete* attribute. Values can be:

NO

Not a non-roll delete message.

YES

Message is non-roll delete and can be deleted only by a *delete operator message* (DOM) instruction.

OPER

If the message is non-roll delete but is deleted only by the cursor delete function from an OCS window.

&ZMNRDRET

Indicates whether the message has been received as a result of a NRDRET command being issued by the user. This allows an NCL procedure to detect redisplayed messages and ignore them for event analysis purposes. Value is YES or NO.

&ZMOSRCID (*stem*.PREFIX.ORIGMSGID.ID)

Contains the message prefix of the originator of the message just received.

&ZMOSRCTP (*stem*.PREFIX.ORIGMSGID.TYPE)

Indicates the *type* of the originator of the message just received. Values can be:

null

If the message was generated within this system.

ROF

If the message was delivered across a ROF session.

MAIOC

If the message was delivered across an MAI OC session.

&ZMPPODTA

Indicates whether any PPO message profile information is available concerning this message. Value is YES or NO. If YES, then other message profile variables are available containing information relating to certain PPO attributes of the message.

&ZMPPOMSG

Indicates whether the message originated from PPO. Value is YES or NO.

&ZMPPOSEV (*stem*.MSGATTR.SEVERITY)

If &ZMPPODTA = YES, then the severity level of the PPO message is available in this variable. Value can be U (undeliverable), I (information), W (warning), N (normal), or S (severe).

&ZMPPOTM (*stem*.SOURCE.TIME.HHMMSSTTT)

If &ZMPPODTA = YES, this variable contains the time that the message originated. Time is in the format *HH.MM.SS.TTT*.

&ZMPPOVNO (*stem*.PPOCNTL.VTAMNUM)

If &ZMPPODTA = YES, this variable contains the VTAM message number of the PPO message.

&ZMPREFXD

Indicates whether the message text has been prefixed with identifier values, for example MAI OC session identifier or ROF message prefix. Values are YES or NO.

&ZMPTEXT

Is set to the entire message text, prefixed with any ROF or MAI OC session identifiers according to the current profile settings, as it will appear on the OCS window if the message is allowed to flow to the window unchanged.

&ZMREQID

If &ZINTYPE=REQ (&INTREAD satisfied by a request message), this variable is set to the user ID of the user that issued the INTQUE command that generated the message, or to the NCL ID of the NCL process that issued the INTQUE command or the &WRITE statement. This variable is dependent on &ZINTYPE for relevance, and its setting is categorized by the &ZMREQSRC variable which will also be set. (This is available to &INTREAD only.)

&ZMREQSRC

If &ZINTYPE=REQ (&INTREAD satisfied by a request message), this variable indicates whether the process that generated the message was a user, another NCL process or a system notification. Values are USER, NCL or SYSTEM respectively. (This is available to &INTREAD only.)

&ZMSOLIC

Indicates whether the message was solicited or unsolicited. A solicited message is usually a command response. Values are YES (solicited) or NO.

&ZMSOURCE

Indicates the verb that last set the values of the message profile variables. The suite of message profile variables remain set until changed by the execution of another verb that modifies the suite. Values are:

INTREAD
LOGREAD
MSGREAD

&ZMTEXT (*stem.*TEXT)

Contains the text of the message received. Note that after &LOGREAD this does not include the standard log message heading information of user ID, time and terminal name. These values are available from other message profile variables that are set after &LOGREAD. Values are:

- Message text if the message is a standard text message
- Request message (that is, if &ZMTYPE=MSG or REQ)
- Null if the message is a *delete operator message* instruction (DOM)

&ZMTYPE

Specifies the type of message received after execution of the read verb. Values are:

MSG

The message is a standard text message.

DOM

The message is a delete operator message instruction.

REQ

The message is a *request* message that has satisfied &INTREAD TYPE=ANY or TYPE=REQ.

The &INTREAD Message Profile

The message profile variables set following &INTREAD are as follows. Depending on the setting of certain key variables, some profile variables can be null.

&ZINTYPE

Specifies whether an &INTREAD operation has been satisfied by a *request message* or a *response message*. Values are REQ (request message) or RESP (response message) or NONE (no message).

&ZMALARM

Indicates whether the message will cause the terminal alarm to sound. Value is YES or NO, or null if &ZINTYPE=REQ.

&ZMALLMSG

Indicates whether the message was generated by a MSG ALL command. Value is YES or NO, or null if &ZINTYPE=REQ.

&ZMCOLOUR

Indicates the color attribute of the message. Value can be any one of NONE, RED, BLUE, GREEN, YELLOW, TURQUOISE, PINK, or WHITE. This attribute is null if &ZINTYPE=REQ.

&ZMDOM

Indicates whether the message is a *delete operator message* instruction. Value is YES or NO. This value is dependent also on the setting of the &ZMTYPE variable.

&ZMDOMID

Contains the *delete operator message identifier* (domid) of the message read, if the message has the *non-roll delete* message attribute, as determined by the setting of the &ZMNRD variable. If &ZMNRD=NO or &ZINTYPE=REQ, this variable is set to null.

&ZMEVONID

Is set when the incoming message was generated by an &EVENT verb and contains the *nclid* of the procedure which issued the &EVENT.

&ZMEVPROF

Is set for incoming event messages (N00102) and represents the EDS profile name which resulted in delivery of the event notification.

&ZMEVRCDE

Contains the *routecde* of the incoming event message (N00102), if the ROUTECDE operand was specified on the originating &EVENT verb.

&ZMEVTIME

Is set for incoming event messages (N00102) to the time the event originated. Time is in the format *HH.MM.SS.TTT*.

&ZMEVUSER

Is set when the incoming message was generated by an &EVENT verb and contains the *userid* of the user who issued the &EVENT verb. This variable is also set for some system events and represents the user who was responsible for the event generation.

&ZMFTSMSG

Indicates whether the message originated from the File Transmission Services (FTS) optional feature. Value is YES or NO, or null if &ZINTYPE=REQ.

&ZMHLIGHT

Indicates the display highlighting attributes of the message. Values are NONE, USCORE, REVERSE, or BLINK. This attribute is null if &ZINTYPE=REQ.

&ZMINTENS

Indicates the display intensity attribute of the message. Values are HIGH or LOW or null if no message being processed. This attribute is null if &ZINTYPE=REQ.

&ZMLNODE

Always null.

&ZMLOGCMD

Always null.

&ZMLSRCID

Contains the message prefix of the last handler of the message just received.

&ZMLSRCTP

Indicates the *type* of the last handler of the message just received. Values can be:

Null

If the message was generated within this system.

ROF

If the message was delivered across a ROF session.

MAIOC

If the message was delivered across an MAI OC session.

This attribute is null if &ZINTYPE=REQ.

&ZMLTIME
Always null.

&ZMLUSER
Always null.

&ZMAPNAME
Contains the mapname of the object in $MSG.USERMDO for the current message (if present).

&ZMMONMSG
Indicates whether the message received is a monitor class message. Values are YES or NO. This attribute is null if &ZINTYPE=REQ.

&ZMMSG
Indicates whether the message received is a standard message or not. The setting of this variable is always opposite to the setting of the &ZMDOM variable and is dependent on the setting of the &ZMTYPE variable. This attribute is null if &ZINTYPE=REQ.

&ZMNRD
Indicates whether the message carries the *non-roll delete* attribute. Values can be:

NO
Not a non-roll delete message.

YES
Message is non-roll delete and can be deleted only by a *delete operator message* (DOM) instruction.

OPER
If the message is non-roll delete but is deleted only by the cursor delete function from an OCS window.

&ZMOSRCID
Contains the message prefix of the originator of the message just received. Will be null if &ZINTYPE=REQ.

&ZMOSRCTP
Indicates the *type* of the originator of the message just received. Values can be:

Null
If the message was generated within this system.

ROF
If the message was delivered across a ROF session.

MAIOC

If the message was delivered across an MAI OC session.

This attribute is null if &ZINTYPE=REQ.

&ZMPREFXD

Indicates whether the message text has been prefixed with identifier values, for example MAI OC session identifier or ROF message prefix. Values are YES or NO. This attribute is null if &ZINTYPE=REQ.

&ZMPTEXT

Is set to the entire message text, prefixed with any ROF or MAI OC session identifiers according to the current profile settings, as it will appear on the OCS window if the message is allowed to flow to the window unchanged. Will be null if &ZINTYPE=REQ.

&ZMREQID

If &ZINTYPE=REQ (&INTREAD satisfied by a request message), this variable is set to the user ID of the user that issued the INTQUE command that generated the request message, or to the NCL ID of the NCL process that issued the INTQUE command. This variable is dependent on &ZINTYPE for relevance, and its setting is categorized by the &ZMREQSRC variable which will also be set. (This is available to &INTREAD only.)

This attribute is null if &ZINTYPE=RESP.

&ZMREQSRC

If &ZINTYPE=REQ (&INTREAD satisfied by a request message), this variable indicates whether the source of the INTQUE command that generated the message was a user or another NCL process. Values are USER or NCL respectively. (This is available to &INTREAD only.)

This attribute is null if &ZINTYPE=RESP.

&ZMSOLIC

Indicates whether the message was solicited or unsolicited. A solicited message is usually a command response. Values are YES (solicited) or NO. This attribute is null if &ZINTYPE=REQ.

&ZMSOURCE

Indicates the verb that last set the values of the message profile variables. It will always be INTREAD for the &INTREAD message profile. The suite of message profile variables remain set until changed by the execution of another verb that modifies the suite.

&ZMTEXT

Contains the text of the message received. This attribute is null if &ZMTYPE=DOM.

&ZMTYPE

Specifies the type of message received after execution of the read verb. Values are:

MSG

The message is a standard text message.

DOM

The message is a delete operator message instruction.

REQ

The message is a *request* message that has satisfied &INTREAD TYPE=ANY or TYPE=REQ.

The &LOGREAD Message Profile

The message profile variables set following &LOGREAD are as follows. Depending on the setting of certain key variables, some profile variables can be null.

&ZMALARM
Indicates whether the message will cause the terminal alarm to sound. Value is YES or NO.

&ZMALLMSG
Indicates whether the message was generated by a MSG ALL command. Value is YES or NO.

&ZMCOLOUR
Indicates the color attribute of the message. Value can be any one of NONE, RED, BLUE, GREEN, YELLOW, TURQUOISE, PINK, or WHITE.

&ZMDOM
Value is always NO.

&ZMDOMID
Contains the *delete operator message identifier* (domid) of the message read, if the message has the *non-roll delete* message attribute, as determined by the setting of the &ZMNRD variable. If &ZMNRD=NO, this variable is set to null.

&ZMFTSMSG
Indicates whether the message originated from the File Transmission Services (FTS) optional feature. Value is YES or NO.

&ZMHLIGHT
Indicates the display highlighting attributes of the message. Values are NONE, USCORE, REVERSE, or BLINK.

&ZMINTENS
Indicates the display intensity attribute of the message. Values are HIGH or LOW or null if no message being processed.

&ZMLNODE
Indicates the terminal name of the user to whom a log message is to be attributed. Value is the name of a terminal. (This is available to &LOGREAD only.)

&ZMLOGCMD
Indicates whether a log message is an echo to the log of a command. Value is YES or NO. (This is available to &LOGREAD only.)

&ZMLSRCID

Contains the message prefix of the last handler of the message just received.

&ZMLSRCTP

Indicates the *type* of the last handler of the message just received. Values can be:

Null

If the message was generated within this system.

ROF

If the message was delivered across a ROF session.

MAIOC

If the message was delivered across an MAI OC session.

&ZMLTIME

Contains the timestamp of a log message. Time is in the format *HH.MM.SS.TTT.*

&ZMLUSER

Contains the user ID to whom generation of the log message is to be attributed.

&ZMAPNAME

Contains the mapname of the object in $MSG.USERMDO for the current message (if present).

&ZMMONMSG

Indicates whether the message received is a monitor class message. Values are YES or NO.

&ZMMSG

Value is always YES.

&ZMNRD

Indicates whether the message carries the *non-roll delete* attribute. Values can be:

NO

Not a non-roll delete message.

YES

Message is non-roll delete and can be deleted only by a *delete operator message* (DOM) instruction.

OPER

If the message is non-roll delete but is deleted only by the cursor delete function from an OCS window.

&ZMOSRCID

Contains the message prefix of the originator of the message just received.

&ZMOSRCTP

V the *type* of the originator of the message just received. Values can be:

Null

If the message was generated within this system.

ROF

If the message was delivered across a ROF session.

MAIOC

If the message was delivered across an MAI OC session.

&ZMPREFXD

Indicates whether the message text has been prefixed with identifier values, for example MAI OC session identifier or ROF message prefix. Values are YES or NO.

&ZMPTEXT

Value is always null.

&ZMREQID

Value is always null.

&ZMREQSRC

Value is always null.

&ZMSOLIC

Indicates whether the message was solicited or unsolicited. A solicited message is usually a command response. Values are YES (solicited) or NO.

&ZMSOURCE

Indicates the verb that last set the values of the message profile variables. This will always be LOGREAD for the &LOGREAD message profile. The suite of message profile variables remain set until changed by the execution of another verb that modifies the suite.

&ZMTEXT

Contains the text of the message received. Note that after &LOGREAD this does not include the standard log message heading information of user ID, time and terminal name. These values are available from other message profile variables that are set after &LOGREAD.

Note that the text does not include the standard log message header of user ID, time and terminal ID.

&ZMTYPE

Value is always MSG.

The &MSGREAD Message Profile

The message profile variables set following &MSGREAD are as follows. Depending on the setting of certain key variables, some profile variables can be null.

&ZMALARM
Indicates terminal alarm. Values are YES or NO.

&ZMALLMSG
Indicates whether the message was generated by a MSG ALL command. Value is YES or NO, or null if &ZMTYPE=DOM.

&ZMCOLOUR
Indicates the color attribute of the message. Value can be any one of NONE, RED, BLUE, GREEN, YELLOW, TURQUOISE, PINK, or WHITE. This attribute is null if &ZMTYPE=DOM.

&ZMDOM
If &ZMTYPE=DOM, value is DOM; otherwise, it is NO.

&ZMDOMID
Contains the *delete operator message identifier* (domid) of the message read, if the message has the *non-roll delete* message attribute, as determined by the setting of the &ZMNRD variable. If &ZMNRD=NO, this variable is set to null.

&ZMFTSMSG
Indicates whether the message originated from the File Transmission Services (FTS) optional feature. Value is YES or NO. This attribute is null if &ZMTYPE=DOM.

&ZMHLIGHT
Indicates the display highlighting attributes of the message. Values are NONE, USCORE, REVERSE, or BLINK. This attribute is null if &ZMTYPE=DOM.

&ZMINTENS
Indicates the display intensity attribute of the message. Values are HIGH or LOW or null if no message being processed. This attribute is null if &ZMTYPE=DOM.

&ZMLOGCMD
Value is always null.

&ZMLNODE
Value is always null.

&ZMLSRCID

Contains the message prefix of the last handler of the message just received. Values can be:

Null

Message generated internally by this system.

ROF *msgid*

Message prefix of the ROF session which delivered the message.

MAIOC *sessionid*

The session ID of the MAI OC session that delivered the message.

This attribute is null if &ZMTYPE=DOM.

&ZMLSRCTP

Indicates the *type* of the last handler of the message just received. Values can be:

Null

If the message was generated within this system.

ROF

If the message was delivered across a ROF session.

MAIOC

If the message was delivered across an MAI OC session.

This attribute is null if &ZMTYPE=DOM.

&ZMLTIME

Value is always null.

&ZMLUSER

Value is always null.

&ZMAPNAME

Contains the mapname of the object in $MSG.USERMDO for the current message (if present).

&ZMMONMSG

Indicates whether the message received is a monitor class message. Values are YES or NO.

&ZMMSG

Value is YES or NO.

&ZMNRD

Indicates whether the message carries the *non-roll delete* attribute. Values can be:

NO

Not a non-roll delete message.

YES

Message is non-roll delete and can be deleted only by a *delete operator message* (DOM) instruction.

OPER

If the message is non-roll delete but is deleted only by the cursor delete function from an OCS window.

&ZMOSRCID

Contains the message prefix of the originator of the message just received. Will be null if &ZMTYPE=DOM.

&ZMOSRCTP

Indicates the *type* of the originator of the message just received. Values can be:

Null

If the message was generated within this system.

ROF

If the message was delivered across a ROF session.

MAIOC

If the message was delivered across an MAI OC session.

This attribute is null if &ZMTYPE=DOM.

&ZMPREFXD

Indicates whether the message text has been prefixed with identifier values, for example MAI OC session identifier or ROF message prefix. Values are YES or NO. This attribute is null if &ZMTYPE=DOM.

&ZMPTEXT

Is set to the entire message text, prefixed with any ROF or MAI OC session identifiers according to the current profile settings, as it will appear on the OCS window if the message is allowed to flow to the window unchanged. This attribute is null if &ZMTYPE=DOM.

&ZMREQID

Value is always null.

&ZMREQSRC

Value is always null.

&ZMSOLIC

Indicates whether the message was solicited or unsolicited. A solicited message is usually a command response. Values are YES (solicited) or NO. This attribute is null if &ZMTYPE=DOM.

&ZMSOURCE

Indicates the verb that last set the values of the message profile variables. This will always be MSGREAD for the &MSGREAD message profile. The suite of message profile variables remain set until changed by the execution of another verb that modifies the suite.

&ZMTEXT

If &ZMTYPE=MSG, &ZMTEXT contains the entire message text. If &ZMTYPE=DOM, then &ZMTEXT is null.

&ZMTYPE

Specifies the type of message received after execution of the read verb. Values are:

MSG

The message is a standard text message.

DOM

The message is a delete operator message instruction.

REQ

The message is a *request* message that has satisfied &INTREAD TYPE=ANY or TYPE=REQ.

The value DOM is possible only if DOM=YES is coded on the &MSGREAD statement.

The &PPOREAD Message Profile

The message profile variables set following &PPOREAD are private to the PPOPROC system procedure, and are different from the variables that form the message profile for the &INTREAD, &LOGREAD and &MSGREAD verbs.

&PPOALERT

Value is YES if this message was routed to PPOPROC as a result of a &PPOALERT verb. &PPOALERT can be used to send messages to PPOPROC either locally or in remote systems. If the message was not originated by a &PPOALERT then value is NO.

&PPOCOLOR

Value is set according to the SYSPARMS PPOCOLOR= operand.

&PPODEFM

Value is YES if the message number has been defined by the DEFMSG command for delivery to PPOPROC. This variable can be set to NO if delivery for the UNSOLICITED class of messages PPOPROC delivery IS set but specific delivery for this message number is NOT.

&PPODOMID

The delete operator message identifier (DOMID) of the message if the PPO message is non-roll delete. Value is DOMID if &PPONRD = YES; otherwise, this variable is set to null value.

&PPODLOC

Value is YES if the message number is one that has been defined through the DEFMSG command for LOCAL delivery. Value is NO if message has not been defined for delivery to local receivers.

&PPODREM

Value is YES if the message number is one that has been defined through the DEFMSG command for REMOTE delivery. Value is NO if message has not been defined for delivery to remote receivers.

&PPOFIRST

Indicates the first message from a possible multi-line message block delivered to PPOPROC. The purpose of this variable is to indicate a new VTAM message group but not necessarily the first message of the group delivered by VTAM as these might not be eligible for delivery to PPOPROC.

&PPOHLITE

Value is set according to the SYSPARMS PPOHLITE= operand.

&PPOLDID

Domain ID of the last SOLVE system to handle this PPO message. If generated by the local system, this variable will contain the domain ID of the local system. The value can be different from the originating domain ID if the message was onward routed by an intermediate system.

&PPOMSGNO

The VTAM message number of the PPO message just received. Note that this value is also available in the &ZMPPOVNO message profile variable following a &INTREAD, &LOGREAD, or &MSGREAD operation.

&PPOMSGSV

The VTAM message severity level associated with the PPO message just received. Value can be U (undeliverable), I (information), W (warning), N (normal), or S (severe). Note that this value is also available in the &ZMPPOSEV message profile variable following a &INTREAD, &LOGREAD, or &MSGREAD operation.

&PPONRD

Value is YES if the PPO message has the non-roll delete attribute (which occurs if a reply is required to the message). Otherwise, value is NO.

&PPOODID

The domain ID of the SOLVE system from which the PPO message originated.

&PPOONETN

The network name of the VTAM that generated the PPO message.

&PPOONMID

The Management Services ID (from the SYSPARMS ID= operand) of the system that generated the message.

&PPOPRIRN

If the PPO message contains qualified network resource names this is the primary NETWORK name.

Note

Management Services analyzes message text based on VTAM tables to extract resource names. This extraction does not necessarily reflect SNA class resources, and can vary with the version of VTAM.

We recommend that resource extraction be performed at message level by PPOPROC, as the Management Services extracted values might not reflect the true resource hierarchy in the VTAM multi-line messages.

&PPOPRIRS

If the PPO message contains network resource names this is the primary resource name. In the context of PPOPROC, this is the resource name of the first message in a VTAM multi-line display.

Note

Management Services analyzes message text based on VTAM tables to extract resource names. This extraction does not necessarily reflect SNA class resources, and can vary with the version of VTAM.

We recommend that resource extraction be performed at message level by PPOPROC, as the Management Services extracted values might not reflect the true resource hierarchy in the VTAM multi-line messages.

&PPOSECRS

If the PPO message contains network resource names this is the secondary resource name. If the message is part of a VTAM multi-line message, this is the first resource name in the current message.

&PPOSECRN

If the PPO message contains qualified network resource names, this is the secondary NETWORK name.

&PPOOSSCP

The SSCP name of the VTAM that generated the PPO message.

&PPOTEXT

The PPO message text.

&PPOTIME

The PPO message generation time in *HH.MM.SS.TTT* format. For messages delivered across an ISR link it reflects time from the remote system converted to the time on the local system.

&PPOTYPE

Defines the type of PPO message received. Value is PPO if the message is a standard unsolicited message delivered from VTAM, SPO if the message is actually the reply to a command issued by an OCS operator or NCL procedure or CMD if the message is a copy of a command entered by an OCS operator or NCL procedure.

&PPOUID

The user ID associated with a message or command which has &PPOTYPE = CMD or &PPOTYPE = SPO, or for which &PPOALRT = YES.

&PPOVMSG

Indicates (YES or NO) whether a message was found as a VTAM message in the DEFMSG table.

E

Sample APPC Conversations

This appendix explains how to use the sample APPC conversations provided in the distribution library. It provides a description of the sample conversations that run between two SOLVE systems.

For a detailed discussion of Management Services APPC programming facilities, see Chapter 11, *Using Advanced Program-to-Program Communication (APPC)*, and Chapter 12, *SOLVE APPC Extensions*, in this manual, and the chapter on APPC in the *Management Services Implementation and Administration Guide.*

Sample Conversations Between Two SOLVE Systems

The purpose of the sample APPC conversations is to illustrate the use of the &APPC NCL verb. The samples are provided in the distribution library. A complete list of sample conversations and detailed instructions on how to run them are also given in the member $SAAPDOC.

Source and Target NCL Procedures

When two NCL procedures communicate using an APPC connection, the one requesting the connection (via the &APPC ALLOCATE verb) is called the *source* procedure. As a result of the allocation request, a target procedure is started or *attached* in the remote system. All the sample conversations involve a source and a target procedure pair.

To allocate a conversation, the source procedure issues an &APPC ALLOCATE verb specifying the appropriate transaction identifier. In the sample conversations, source and target NCL procedure pairs are denoted by $SAAPS*xx* and $SAAPT*xx* respectively and the associated transaction identifier by $SATRN*xx*.

Running the Sample APPC Conversations

To run the sample APPC conversations, choose one or more of the transactions available ($SATRN*xx*) and one of the environments described below. The sample conversations are assumed to run between a *source* SOLVE system (NMA) and a *target* SOLVE (NMB). If you want to run the samples, you should replace these by the LU names of your particular SOLVE systems.

Running the APPC samples involves the following steps:

- APPC environment definitions. This involves defining transactions and APPC links to both the source and the target systems. Examples of three different environments are provided below.
- Running the APPC transactions. This involves starting the source procedure $SAAPS*xx* in the source system NMA. Information about the completion of &APPC verbs (from &RETCODE and &ZFDBK), the ‘what-received’ indicator (from &ZAPPCWR) and the state of the conversation (from &ZAPPCSTA) is reported in messages written to the system log. A description of the first three sample conversations is given below.

Environment 1: Local Conversations

This environment involves only one SOLVE system. Both the source procedure $SAAPS*xx* and the target procedure $SAAPT*xx* run in the same system and the same NCL region. Since no LU6.2 sessions are established no link definitions are necessary. Environments 1 and 2 are the simplest in which to run the sample conversations and require only transaction definitions, as follows:

```
DEFTRANS TRANSID=$SATRNxx PROC=$SAAPTxx
```

The default destination information for the transaction is omitted, causing the ENV=CURRENT parameter to be assumed, which indicates that $SAAPT*xx* is to be started in the same SOLVE system and the same NCL region as the source procedure.

Environment 2: Same LU conversations

This environment involves only one SOLVE system. Both the source procedure $SAAPS*xx* and the target procedure $SAAPT*xx* run in the same system. In this case, however, the target procedure is started in the background server environment, BSVR. As no LU6.2 sessions are established, no link definitions are necessary. Environments 1 and 2 are the simplest in which to run the sample conversations and require only transaction definitions, as follows:

```
DEFTRANS TRANSID=$SATRNxxLU=NMA PROC=$SAAPTxx
```

The operand LU=NMA indicates that $SAAPT*xx*is to be started in the same SOLVE system as the source procedure.

Environment 3: Conversations Between Two SOLVE Systems

In this environment, the source procedure, $SAAPS*xx*, is started in NMA and the target procedure, $SAAPT*xx*, is started in NMB. A single session APPC link is established between NMA and NMB and both link and transaction definitions are needed.

Definitions in the source system NMA:

```
DEFTRANS TRANSID=$SATRNxxLINK=NMB
DEFLINK TYPE=APPC LINK=NMB LU=NMB MON=YES
```

Definitions in the target system NMB:

```
DEFTRANS TRANSID=$SATRNxxPROC=$SAAPTxx
DEFLINK TYPE=APPC LINK=NMA LU=NMA
```

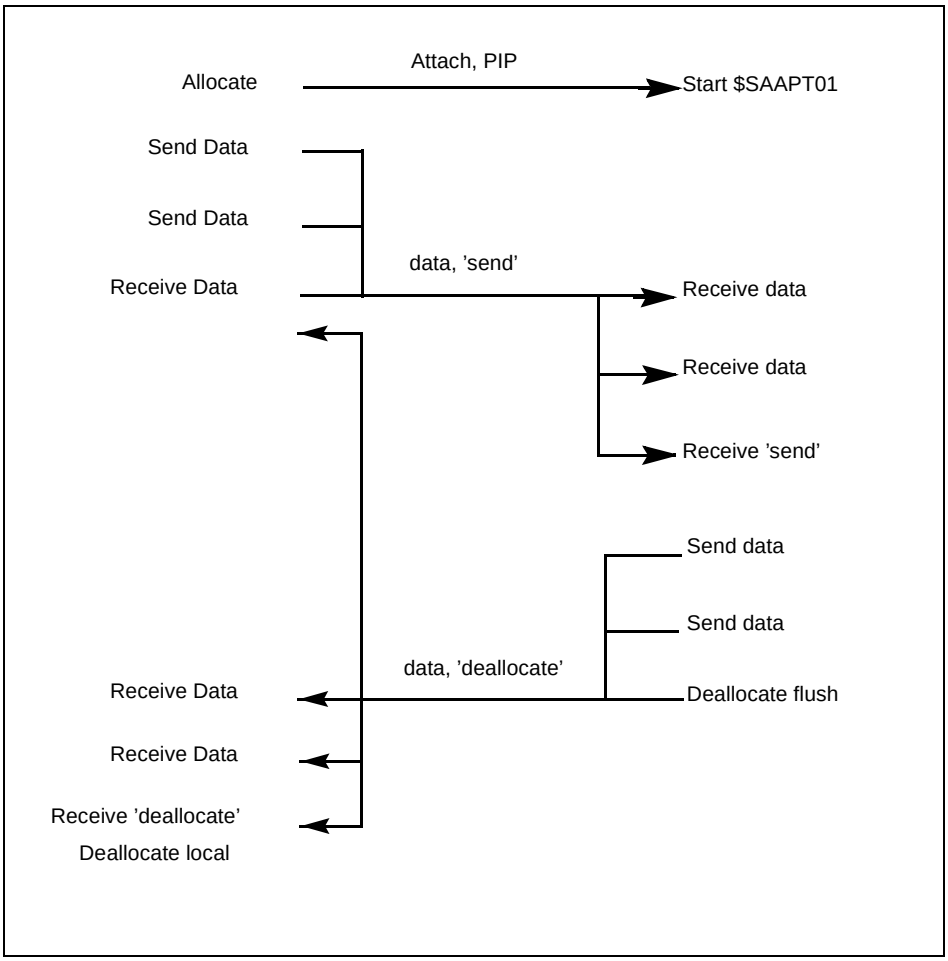
Sample APPC Conversations: Description

Following is a description of the first three sample APPC conversations which are also illustrated in Figure E-1, Figure E-2, and Figure E-3. These are:

- Transaction $SATRN01: using RECEIVE_AND_WAIT to change direction.
  Source procedure $SAAPS01, target procedure $SAAPT01.
- Transaction $SATRN02: using CONFIRM/CONFIRMED
  Source procedure $SAAPS02, target procedure $SAAPT02.
- Transaction $SATRN03: using SEND_ERROR
  Source procedure $SAAPS03, target procedure $SAAPT03.

Using RECEIVE_AND_WAIT to Change Direction

Figure E-1. Sample Transaction $SATRN01



The following discussion refers to Figure E-1, which shows the conversation flow for transaction $SATRN01.

This example illustrates the use of RECEIVE_AND_WAIT to change the direction of data flow. The source procedure $SAAPS01 issues an allocation request for transaction $SATRN01. This starts the target procedure, $SAAPT01, in the remote system. PIP data is also sent as part of the attach request. The source procedure then issues SEND_DATA verbs which cause data to accumulate in the output buffers.

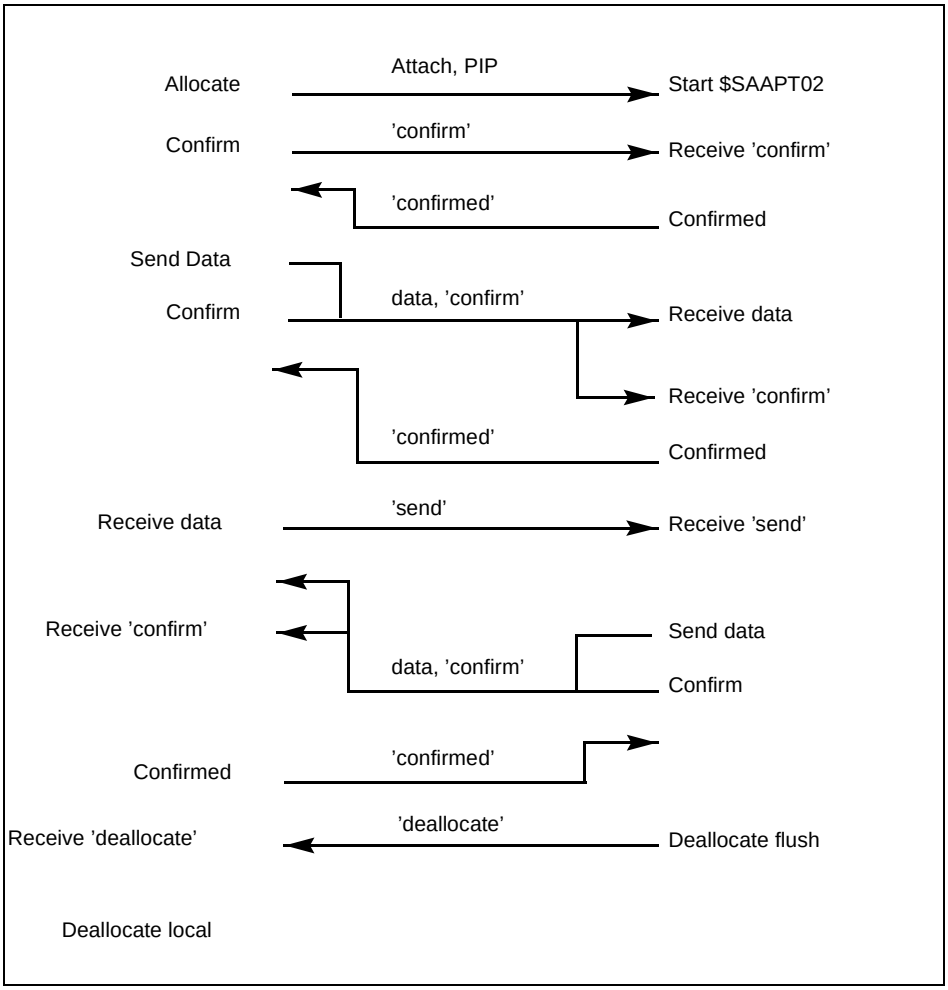
Issuing RECEIVE_AND_WAIT from SEND state forces the source side of the conversation into RECEIVE state and the transmission of all data accumulated in the output buffers. The SEND indicator is also transmitted which notifies the other side to start sending data. The target procedure issues RECEIVE_AND_WAIT to receive data (&ZAPPCWR set to DATA_COMPLETE) and to receive the SEND indicator (&ZAPPCWR set to SEND) which causes it to enter SEND state.

When the target procedure finishes sending its data, it deallocates the conversation with the FLUSH option which causes the transmission of all data in the output buffers, including the DEALLOCATE indicator.

The source procedure issues RECEIVE_AND_WAIT until it receives DEALLOCATE in &ZAPPCWR (&RETCODE = 4 and &ZFDBK = 0) and enters DEALLOCATE state. It then deallocates the conversation locally by issuing DEALLOCATE with the LOCAL option.

Using CONFIRM/CONFIRMED

Figure E-2. Sample Transaction $SATRN02



The following discussion refers to Figure E-2 which shows the conversation flow for transaction $SATRN02. This example illustrates the use of CONFIRM/CONFIRMED verbs to synchronize the sending and receiving of data.

The first CONFIRM/CONFIRMED pair is used to ensure that the target procedure, $SAAPT02, has been started before proceeding with the conversation. The RECEIVE_AND_WAIT issued by the target procedure sets the &ZAPPCWR to CONFIRM and causes it to enter CONFIRM state. The CONFIRM verb issued by the source procedure will not complete until CONFIRMED is received from $SAAPT02.

The second CONFIRM/CONFIRMED pair is used to synchronize the sending and receiving of data. The source procedure issues SEND_DATA followed by CONFIRM, which causes all data to be transmitted, including the CONFIRM indicator.

The target procedure issues two RECEIVE_AND_WAIT verbs, the first gets the data, the second gets the CONFIRM indicator (in &ZAPPCWR) forcing it to enter CONFIRM state. It then issues CONFIRMED, which returns it to RECEIVE state and causes the transmission of the CONFIRMED indicator. The CONFIRM verb in the source side completes with &RETCODE = 0.

The direction of data flow is changed by the source procedure issuing RECEIVE_AND_WAIT from SEND state. The CONFIRM/CONFIRMED process is then repeated before the conversation is deallocated.

Using SEND_ERROR

The following discussion refers to Figure E-3, which shows the conversation flow for transaction $SATRN03. In this example, the SEND_ERROR verb is used to notify the sending procedure that the data received is invalid.

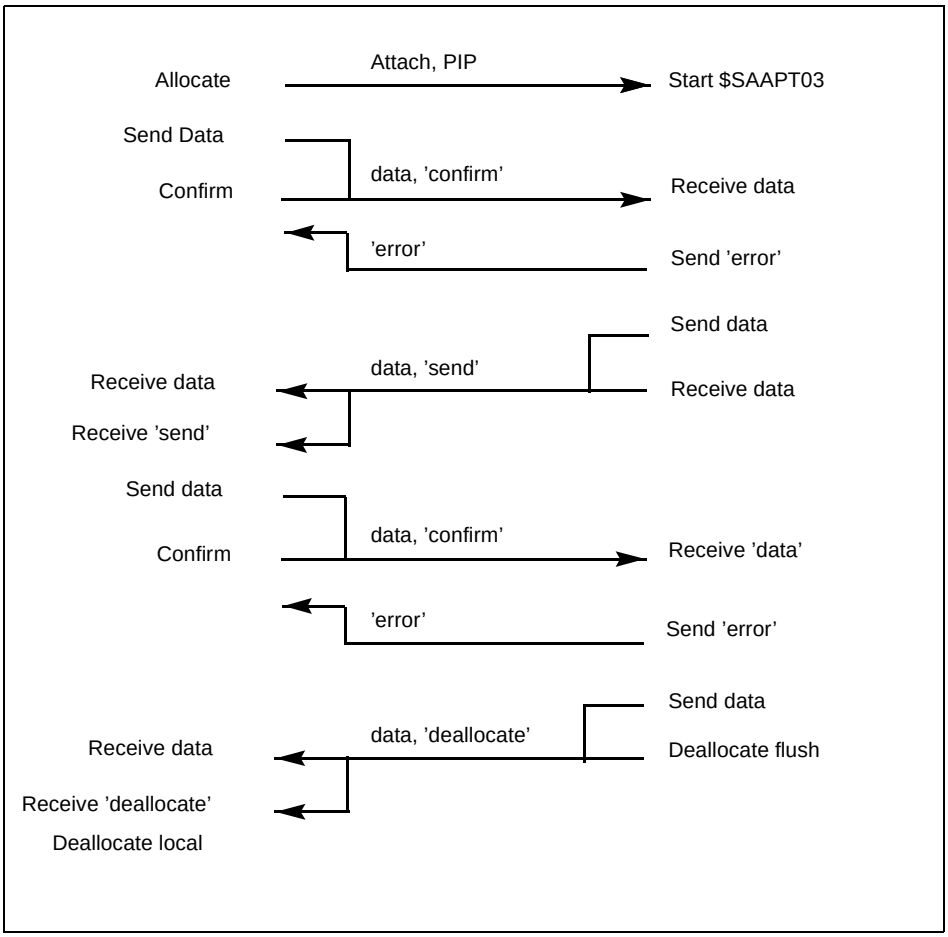
The source procedure starts by issuing SEND_DATA followed by CONFIRM. This causes the data and the CONFIRM indicator to be transmitted.

To notify the sending procedure that the data received is invalid, the target procedure issues SEND_ERROR from RECEIVE state. This causes it to enter SEND state and to transmit an ERROR indication. This in turn causes the CONFIRM verb issued by the source side to complete with &RETCODE = 8 and to enter RECEIVE state.

This change of direction allows the target procedure to send an indication of the error encountered by issuing a SEND_DATA verb with appropriate error information. It then issues RECEIVE_AND_WAIT to resume the process of receiving data.

The remaining verbs illustrate the use of SEND_ERROR from CONFIRM state. In this case however after sending information about the error the target procedure terminates the conversation by issuing DEALLOCATE with the FLUSH option.

Figure E-3. Sample Transaction $SATRN03



F

NDB Response Codes

This appendix lists the response codes that can be returned by the NDB command, and &NDB statements.

The response codes fall into three categories:

- All OK (response 0)
- Warning conditions (1 to 29)
- Error conditions (30 to 255)

An NCL procedure will be terminated if an error response is returned, unless &NDBCTL ERROR=CONTINUE is in effect, except as noted below:

- &NDBOPEN is always processed as if &NDBCTL ERROR=CONTINUE is in effect, except for response 34 (already open), which honors the actual setting of &NDBCTL ERROR.
- &NDBCLOSE is always processed as if &NDBCTL ERROR=CONTINUE is in effect, except for response 35 (not open), which honors the actual setting of &NDBCTL ERROR.

These exceptions allow a procedure to recover from any error conditions encountered when opening or closing an NDB, except for the obvious programming error (already open or closed). This should encourage you to develop exception handling logic when first writing a procedure.

If an error response from an &NDBOPEN is ignored, the next &NDB statement referring to that NDB will cause the procedure to abnormally terminate (as a result of trying to access an NDB that has not been opened).

Error Information

Each response code description includes an indication of the information that will be provided in the &NDBERRI system variable. N/A means there is no information provided, and &NDBERRI is null.

Response Codes

The response codes are listed below, in ascending order:

0 ERRI=N/A
Database processing completed with no errors.

1 ERRI=N/A
Record with requested key or RID not found (GET/UPD/DEL), OR reposition RID not in scan result list (SEQ RESET).

2 ERRI=N/A
End of file on GET sequential (forward or backward).

3 ERRI=field name
Named field not found for INFO on field name.

4 ERRI=N/A
No records found for supplied SCAN criteria.

5 ERRI=N/A
SCAN terminated—exceeded I/O limit.

6 ERRI=N/A
SCAN terminated—exceeded time limit.

7 ERRI=N/A
SCAN terminated—exceeded storage limit.

8 ERRI=N/A
SCAN terminated—met or exceeded number of records limit.

10-19 ERRI=N/A
Linked &NDBGET/&NDBFMT—extra no find return code as used in the link definition.

20 ERRI=format name
FMT del, format name not found.

21 ERRI=sequence name
SEQ del, sequence not found.

29 ERRI=N/A
Maximum possible warning code—currently not assigned.

30 ERRI=N/A
Error in request (catch-all). See command/messages.

31 ERRI=N/A
Logic error in request: not signed on to database (NDB). For example, if the NCL procedure is paused and the database stopped/started, then the next call gets this error.

32 ERRI=N/A
Insufficient storage to process request.

33 ERRI=N/A
OPEN request rejected, database is stopping.

34 ERRI=N/A
&NDBOPEN—already open to this NDB.

35 ERRI=N/A
&NDBCLOSE—not open to this NDB.

36 ERRI=N/A
NDB PURGE—nominated NDB not found, could not purge.

37 ERRI=N/A
NDB PURGE—nominated NDB not locked, cannot purge.

38 ERRI=N/A
Load for load module for requested NDB function failed.

39 ERRI=N/A
Requested option not allowed—NDB is in load mode.

40 ERRI=N/A
&NDBOPEN failed by NCLEX01 user exit.

41 ERRI=N/A
This &NDB... verb blocked by &NDBOPEN user exit (NCLEX01) return mask.

51 ERRI=N/A
Database open failure: VFS open for database failed.
Possible causes:
- File not allocated to SOLVE.
- UDBCTL OPEN not done or in error.

52 ERRI=N/A
Database open failure: Specified database name is not a VSAM KSDS.

53 ERRI=N/A
Database open failure: VSAM relative key position (RKP) not 0.

54 ERRI=N/A
Database open failure: VSAM key length too short (that is, fewer than 16) to be a valid NDB database.

55 ERRI=N/A
Database open failure: VSAM data length too short to be a valid NDB database.

56 ERRI=N/A
Database open failure: Read of control record failed.

57 ERRI=N/A
Database open failure: Control record not valid.

58 ERRI=N/A
Database open failure: Unable to get storage to contain control record.

59 ERRI=N/A
Database open failure: Read of transaction control record failed.

60 ERRI=N/A
Database open failure: Transaction control record not valid.

61 ERRI=N/A
Database open failure: Unable to get storage to contain transaction control record.

62 ERRI=N/A
Database open failure: Unable to get storage to contain transaction data record.

63 ERRI=N/A
Database open failure: Error building field name indexes.

64 ERRI=N/A
Database open failure: Error re-applying pending transaction. Probable file full condition.

65 ERRI=N/A
Database open failure: Database flagged as in DEFER status. Probable file corrupt condition, system has failed with database started in DEFER mode.

66 ERRI=N/A
Database open failure: UDB is open by other users.

67 ERRI=N/A
. UDB is open INPUT and NDB START or &NDBOPEN is not for INPUT only mode

68 ERRI=N/A
Domain ID on NDB control record and this SOLVE system mismatch, and FORCE not on NDB START. NDB is not started. Warning—it might be currently open in another SOLVE system.

69 ERRI=N/A
NDB version in this NDB control record not supported.

70 ERRI=N/A
This NDB is open under another file ID on this SOLVE (VSAM timestamp match).

71 ERRI=N/A
Request not allowed on active database. For example, NDB CREATE, NDB RESET, NDB ALTER, or NDB UNLOAD.

72 ERRI=N/A
User ID required for request. Internal failure, should never occur with NDB command.

73 ERRI=sequence name
Specified sequence name not defined.

74 ERRI=sequence name
Sequence name required or already defined.

75 ERRI=N/A
Free-format text had an invalid token:
- too long
- unmatched quotes.

76 ERRI=N/A
Logic error—not signed on to database (NDB). For example, if NCL proc is paused, and DB stopped/started, then next call will get this error.

77 ERRI=N/A
This user ID has an asynchronous request running. Internal failure; should never occur with NDB command or &NDB statements.

78 ERRI=N/A
Request internally cancelled.

79 ERRI=N/A
OPEN EXCLUSIVE—other users on database.

80 ERRI=N/A
OPEN—database locked by an exclusive user.

81 ERRI=field name if relevant
Add, update, or delete field, an error in the field list syntax, etc.

82 ERRI=N/A
Add, update, or delete field, an internal error in the field index.

83 ERRI=field name
Value for named field is bad, not recognizable as valid data.

84 ERRI=field name
Value for named field is too long.

85 ERRI=field name
Value for named numeric field is not numeric, or outside range: -2,147,483,648 : 2,147,483,647.

86 ERRI=field name
Value for named hex field is not a valid hex string, or is an odd number of characters.

87 ERRI=field name
Value for named DATE or CDATE field is not valid.

88 ERRI=field name
Field name is not a valid floating point number.

89 ERRI=field name
Supplied value is not a valid hexadecimal number.

90 ERRI=field name
Value for named time field is not a valid time, in the format: HHMMSS or HHMMSS.TTTTTT.

92 ERRI=N/A
Operation not allowed. NDB or user open in input-only mode.

93 ERRI=field name
Field update request—could not read field record from NDB.

94 ERRI=field name
Supplied value is not a valid timestamp, in the format: YYYYMMDDHHMMSS.TTTTTT.

101 ERRI=token in error
ADD or UPD, field=value list syntax error.

102 ERRI=field name
ADD or UPD, required field not provided.

103 ERRI=field name
ADD record, sequence key value already on database (that is, not unique).

104 ERRI=field name
ADD/UPD record, KEY=UNIQUE field value not unique.

105 ERRI=field name
UPD record, UPDATE=NO value change for field.

106 ERRI=format name
Format specified on an ADD or UPD does not exist.

107 ERRI=format name
Format specified on an ADD or UPD exists but is an INPUT format and cannot be used for output.

111 ERRI=format name
FMT add, format name '*' is invalid.

112 ERRI=format name
FMT add, format name already exists.

114 ERRI=token in error
FMT/GET, format list syntax error.

115 ERRI=field name
FMT/GET, field name not defined in database.

116 ERRI=format name
Format specified on a GET exists but is an OUTPUT format and cannot be used for input.

117 ERRI=field name
You have referenced an NDB field twice in an OUTPUT format definition.

118 ERRI=field name
You have used an NCL keyword name in an OUTPUT format, but the keyword is not an NCL system variable name.

121 ERRI=format name
GET, supplied format name not defined.

122 ERRI=field name
GET by key, key field not defined on database.

123 ERRI=field name
GET by key, field not keyed.

124 ERRI=sequence name
GET by sequence, sequence ID not defined.

125 ERRI=sequence name
GET by sequence, key field for sequence has been deleted (by &NDBDEF DELETE).

126 ERRI=sequence name
GET by sequence, skip=0 specified and not currently positioned.

127 ERRI=sequence name
GET by sequence, skip=0 specified and currently at EOF (front or back).

128 ERRI=field name
GET by key field, GENERIC= is invalid for this field format. Generic access is only allowed for character and HEX fields.

130 ERRI=field name
&NDBGET histogram (KEY=) on sequence key not supported.

131 ERRI=sequence name
SEQ DEF—invalid sequence name.

132 ERRI=sequence name
SEQ DEF, sequence name already exists.

133 ERRI=N/A
SEQ DEF, from or to RID invalid.

134 ERRI=sequence name
SEQ DEF, field name not found or is not a key.

135 ERRI=N/A
SEQ DEF, from or to values not valid.

136 ERRI=field name
SEQ DEF, generic invalid with field format.

137 ERRI=N/A
SEQ DEF, invalid null value for GENERIC=.

139 ERRI=sequence name
SEQ RESET, sequence not found.

140 ERRI=invalid data value
SEQ RESET, REPOS= value invalid or null.

141 ERRI=N/A
SEQ RESET, REPOS= not valid on a scan sequence that is not sorted, or, while sorted, the primary sort field was substringed.

142 ERRI=sequence name
SEQ RESET, REPOS by RID only valid for a SCAN sequence.

143 ERRI=N/A
&NDBSEQ histogram (KEY=) on sequence key not supported.

144 ERRI=sequence name
RELPOS sequence is not a sequence constructed by &NDBSCAN.

151 ERRI=N/A
INFO by field number, number *lt* 1 or *gt* number fields in database.

161 ERRI=sequence name
SCAN sequence ID is already defined.

162 ERRI=sequence name
SCAN sequence ID is currently in use, by an active scan for this user.

163 ERRI=field name
SCAN SORT= field name not defined on database.

183 ERRI=N/A
SCAN syntax error in scan request.

191 ERRI=N/A
. Unload failed. Accompanying messages will indicate the cause of the failure

193 ERRI=N/A
NDB ALTER failed.

237 ERRI=N/A
File integrity error, get if continuation DBDR failed. Database possibly corrupted. Contact local Support Center.

238 ERRI=field name
File integrity error, get XFF if DBKR failed. Database possibly corrupted. Contact local Support Center.

239 ERRI=field name
File integrity error, get KGE if DBKR failed. Database possibly corrupted. Contact local Support Center.

240 ERRI=N/A
VSAM I/O Error. See log for more information.

241 ERRI=N/A
Request not processed—invalid request code (internal error), should never occur with NCL (&NDB) or NDB command.

242 ERRI=N/A
Request not processed—RPL busy flag set (internal error), should never occur with NCL (&NDB) or NDB command.

243 ERRI=N/A
Request not processed—insufficient storage to queue request to database handler. Try again.

244 ERRI=N/A
Request not processed—required text parameter not provided (internal error). Should never occur with NCL (&NDB) or NDB command.

245 ERRI=N/A
Request not processed—Database not started.

246 ERRI=N/A
Request not processed—required VAL1 parameter not provided (internal error). Should never occur with NCL (&NDB) or NDB command.

247 ERRI=N/A
Request not processed—required VAL2 parameter not provided (internal error). Should never occur with NCL (&NDB) or NDB command.

248 ERRI=N/A
Request not processed—database not LOCKED. Only applicable to NDB START UNLOCK command.

250 ERRI=N/A
Request not processed—database is LOCKED or STOPPING.

251 ERRI=N/A
Request not processed—long running command currently in progress. For example, NDB UNLOAD.

252 ERRI=abend code
This return code indicates that an NDB operation was terminated due to a logical abend caused by a possible NDB corruption. The log will contain useful debugging information.

For update requests, the NDB will be stopped and locked. For read/scan requests, the request is terminated but the NDB will continue to process other requests.

254 ERRI=N/A
Request not processed—feature not present, or Management Services is shutting down.

G

Using Keyranges with an NDB

If an NDB contains large amounts of data, or if you wish to separate the data records in an NDB from the key and control records, then you might want to use the KEYRANGES AMS parameter.

This appendix provides information on the key structure of an NDB that enables you to do this.

Note

This information is for guidance only. Future releases of Management Services might alter the key structures. If you divide NDBs by key ranges, you should review this appendix whenever a new release of Management Services is installed.

NDB Key Structure

The VSAM key in an NDB is divided into the following parts:

- A type prefix, two bytes long
- For some records, a field code, two bytes long
- Key data, either the rest of the key, or four bytes shorter
- For some keys, a suffix, four bytes long

The following table shows the key structures (in ascending order):

1. Control record (always first record in dataset)

   Rectype:
   X‘0000’

   Rest of key:
   All binary 0

2. Transaction control record (always second record in dataset)

   Rectype:
   X‘0010’

   Rest of key:
   All binary 0

3. Transaction journal records

   Rectype:
   X‘0011’

   Sequence number:
   2 bytes, binary (X‘0001’ to *nnnn*)

   Rest of key:
   All binary 0

4. Field information records (2 types)

   a. First (Basic Field Information):

      Rectype:
      X‘0020’

      Field code:
      2 bytes binary

      Rest of key:
      All binary 0

   b. Second (Contains Key Statistics):

      Rectype:
      X‘0021’

      Field code:
      2 bytes binary

Rest of key:
All binary 0

5. Key records

   Rectype:
   X‘0030’

   Field code:
   2 bytes binary

   Field value:
   (keylen - 8 bytes, see below)

   Suffix:
   4 bytes, binary

6. RID to sequence key records (present only if NDB defined with a sequence key)

   Rectype:
   X‘0032’

   RID:
   4 bytes, binary

   Rest of key:
   All binary 0

7. Data records. There are 2 formats, depending on whether the NDB has a sequence key, or not:

   a. For databases defined without a sequence key, the format is:

      Rectype:
      x‘0040’

      RID
      4 bytes, binary

      Reserved:
      (keylen - 10) bytes long

      Sequence number:
      4 bytes binary (usually 0, used to handle large data records)

   b. For databases defined with a sequence key, the format is:

      Rectype:
      x‘0040’

      field code:
      2 bytes, binary; always x‘0001’

      Seq key value
      (keylen - 8 bytes, see below)

      Sequence number:
      4 bytes binary (usually 0, used to handle large data records)

Storage of Values

Field values and sequence key values are stored as follows:

CHAR fields
The character field value, padded to (keylen-8) with blanks.

NUM fields
4 bytes, binary, with the sign bit inverted (that is, 0 is stored as X‘80000000’, 100 is X‘80000064’). Padded to (keylen-8) with binary 0.

HEX fields
Stored in compressed hex format. Padded to (keylen-9) with binary 0. The last byte (of the value part, not of the total key) contains the significant length in binary (so X‘ABCD’ and X‘ABCD00’ are distinct; X‘ABCD’ is stored as X‘ABCD0000...02’ and X‘ABCD00’ is stored as X‘ABCD0000...03’).

DATE fields
Stored in unsigned packed. For example, 21 September, 1987 is stored as X‘870921’. Padded to (keylen-8) with binary 0.

FLOAT fields
Stored in 8-byte floating point, with the following change to force character compares to work correctly:

- If sign bit is 0 (that is, number is 0 or positive), invert the sign bit.
- If the sign bit is 1 (that is, a negative number), invert the entire 8-byte field.

Key length padding is binary 0.

CDATE fields
Stored internally in 3-byte binary with value 1 (x’000001’) representing 1/1/0001. Key length padding is binary 0.

TIME fields
Stored internally in 5-byte binary as a number of micro-seconds, values from 0 to 86,399,999,999 (86,400 seconds in a day, times 1000000 for microseconds, -1 microsecond). The largest hex value is x’141DD75FFF’. Key length padding is binary 0.

TIMESTAMP fields
Stored internally as a concatenation of the CDATE (3-byte) and TIME (5-byte) fields (CDATE first). Thus a TIMESTAMP takes 8-bytes. Key length padding is binary 0.

Suggested Keyranges

The following are suggested keyrange breakups:

X‘0000’ to X‘002F’

Includes the control, transaction, and field records. The transaction records have the most activity. The field records can be in the same key range, as they are only referenced on NDB START, or when an &NDBDEF statement is processed.

X‘0030’ to X‘003F’

Includes all the key records, and, for databases with a sequence key, the RID to sequence key relation records.

Note

It is not feasible to split the key records by field, as the field code is not readily determined, and can change if an NDB is unloaded and reloaded, if fields have been added or deleted. The only field code which is guaranteed is the field code for a sequence key, which is always X’0001’.

If field codes are really necessary for further splitting of key records, the field information records (record type X‘0020’) contain the field code in bytes 3 to 4 of the key, and also in the data, in the 2 bytes immediately following the key. The next 12 bytes after the field code in the data contain the field name. Remember that this field code can be different if an NDB is unloaded and reloaded.

X‘0040’ to X‘FFFF’

Contains all data records.

For NDBs with a sequence key, the data can be broken on sequence key value by preceding the value limits with X‘00400001’. For example, if an NDB has a character sequence key, and you wish to break A-K, and L-Z, you can use:

X‘00400001C1’ to X‘00400001D2’ (A-K)
X‘00400001D3’to X‘00400001E9’ (L-Z)

Other Considerations

Failure to provide complete key ranges could lead to VSAM errors if a key value cannot be matched to a keyrange.

Using keyranges other than to separate control, key, and data records, or, for NDBs with a sequence key, sequence key ranges, is not recommended.

H

Using NCLEX01 for NDB Security

A user-nominated NCLEX01 exit can be invoked if SYSPARMS NDBOPENX is set to YES. In this case, all &NDBOPEN statements that actually open the NDB (for this process) result in the nominated NCLEX01 exit being called.

NDB Open Exit Call Details

The parameter list passed on an &NDBOPEN call to NCLEX01 is described below:

- The standard parameter list, as mapped by the $NMNCEX1 macro.

  This parameter list contains information common to all calls to NCLEX01 for various reasons. It contains a function code, and information about the current user ID, etc. The security correlator is passed to allow access to any security information provided by any security exits.

  On an &NDBOPEN call, the NEXFUNC field has a value of 12 (decimal).

  Field NEXNDEX1 contains a pointer to a supplementary parameter list containing additional information.

- A supplementary parameter list, mapped by the NEXND DSECT. This DSECT is in the NCLEX01 macro.

  Fields in this DSECT include:

NEXNDNAM	DS CL8	NDB name being opened
NEXNDFL1	DS X	Flag byte
NEXNDOPX	EQU X'80'	... EXCLUSIVE on &NDBOPEN
NEXNDOPI	EQU X'40'	... INPUT on &NDBOPEN
NEXNDOKM	DS 0XL2	Following 2 bytes. Note: initialized to 2X'FF'.
NEXNDOK1	DS X	... first OK operations flag.
NEXNDGTO	EQU X'80'	... 1 = &NDBGET ok.
NEXNDSCO	EQU X'40'	... 1 = &NDBSCAN ok.
NEXNDADO	EQU X'20'	... 1 = &NDBADD ok.
NEXNDUPO	EQU X'10'	... 1 = &NDBUPD ok.
NEXNDDLO	EQU X'08'	... 1 = &NDBDEL ok.
NEXNDDFO	EQU X'04'	... 1 = &NDBDEF ok.
NEXNDINO	EQU X'02'	... 1 = &NDBINFO ok.
NEXNDOK2	DS X	... 2nd ok operations flag.
NEXNDUDL	DS H	Length (0-50 decimal) of user data from &NDBOPEN statement.
NEXNDUDT	DS CL50	User data (padded/truncated to 50 chars) from &NDBOPEN statement (pad is blank).

  These fields can be referenced to determine the action to take.

By returning R15 not equal to 0, all access to the NDB from the NCL process is denied. ASN NDB response code of 40 is returned, and this is handled based on the &NDBCTL ERROR= setting.

If R15 is returned with a 0, the 2-byte NEXNDOKM field is examined. If all the defined bits are off (as listed above), then the effect is the same as setting R15 not 0. Note that the NEXNDOKM field is initialized to all bits on before the call.

If any bits in the NEXNDOKM field are off, the associated &NDB*xxxx* verb cannot be used. Attempts to use it results in an NDB response code of 41. This too is handled as per the &NDBCTL ERROR= setting.

Note that you can deny use of the &NDBDEF verb with this exit. This forces use of the NDB FIELD command to add, update or delete field definitions from the database.

If the exit terminates abnormally while processing the &NDBOPEN request, the NCL process is terminated with an error.

The exit is attached as a subtask in all environments except VSE. It should be coded re-entrant as multiple copies can be executing simultaneously.

Glossary

This glossary defines the terms and abbreviations commonly used with Management Services.

It also includes references to terms used in an IBM environment and any equivalent FUJITSU terms.

3270 VDU terminal
An IBM video display terminal. This is often used to refer to the entire range of 3270 terminals. When followed by a number (for example, 3270-5), a specific model is intended.

370/390
This is an abbreviation for IBM's System 370 or S/370 architecture. It is often used to indicate any mainframe CPU that implements this architecture.

3705/3725/3745/3746
An IBM front end communications processor (The Fujitsu equivalent is a CCP or 2806).

9526
A Fujitsu video display terminal.

ACB (Access method Control Block)
A control block that links an application program to an access method such as IBM's VTAM or VSAM.

ACB-sharing
MAI's ability to use a single VTAM ACB for multiple sessions.

Access Security Exit

An installation-provided routine that may be used to replace the Management Services UAMS functions, partially or completely, allowing logon, logoff, and password maintenance requests to be passed to an external security system.

ACF/VTAM (Advanced Communication Facility/VTAM)

IBM's product implementation of SNA's SSCP or CP.

Activity Log

A system-maintained log that records all important activity for use in later problem determination.

Alternate Index

An alternative view of the data contained within a VSAM keyed dataset. The alternate index allows data to be retrieved using an alternate key in addition to the usual access through the primary key.

AOM (Advanced Operation Management)

A facility of Management Services that manages and controls local and remote operating systems.

AOMPROC

The name given to an NCL procedure used to intercept messages from the screening table component of AOM to provide extended message processing.

APF (Authorized Program Facility)

Describes the special authorization level required within the operating system for certain applications.

APPC (Advanced Program to Program Communications)

An IBM-defined application level protocol which makes use of SNA's LU 6.2. It is an accepted industry standard for transaction processing between peer systems.

APPL (Application Program)

A VTAM term used to describe the definition that allows an application to use VTAM facilities.

APPN (Advance Peer-to-Peer Networking)

IBM's data communications support that routes data in a network between two or more APPC systems that do not need to be adjacent.

Application Plan

A DB2 term that refers to the control structure used by the database to allocate resources and execute SQL statements.

ASN.1

Abstract Syntax Notation One, defined by ISO 8824, is an abstract syntax used to describe data structures. It is used by Mapping Services to define data structures within Management Services.

Authorization ID

A DB2 term that refers to the ID used by DB2 to control access to a database.

BCI (Batch Command Interface)

A subcomponent of EIP that allows commands to be issued from batch jobs into a system.

BER (Basic Encoding Rules)

The transfer syntax used by Mapping Services to serialize data for transmission. It is defined by ISO 8825.

BIND

1. A VTAM term describing the action of logically linking one network resource with another network resource.
2. A DB2 term for the process that connects together the application plan and the Database Request Module (DBRM) used by an application.

Broadcast Services

The message broadcasting function provided by Management Services.

Build/Verify

A semi-automatic method of adding or verifying a series or group of resource records, by retrieving data from the hardware and software in the live configuration.

CAF (Call Attach Facility)

A connection technique provided by IBM and used by the EDBS facility of Management Services, to communicate with DB2.

CAS (Common Application Services)

CAS functions are a collection of NCL routines designed to facilitate program development.

CICS (Customer Information Control System)

An IBM licensed program that enables transactions entered at remote terminals to be processed concurrently by user-written application programs.

Client

A functional unit that receives shared services from a server.

CNM (Communications Network Management)

IBM term for its SNA management facilities.

CNMPROC

The name given to an NCL procedure used to intercept CNM records received across the VTAM CNM interface by the NEWS component of NetMaster for SNA.

Command Partition

A term associated with network partitioning that describes the group of network resources a user ID is authorized to reference with VTAM commands.

Control Member

A term associated with network partitioning that describes the list of resource table names applying to a user ID. This control member is referenced in the definition of USERID.

CP (Control Point)

A collection of tasks which provide directory and route selection functions for APPN.

Cross-domain Resource (CDRSC)

A VTAM term describing the definition of a network resource that is owned by VTAM in another domain.

DBCS (Double Byte Character String)

Refers to a mode of representation of data where each byte of data requires 16 bits rather than 8 bits as required by Single Byte Character String. DBCS is used for the implementation of Asian languages such as Japanese.

Deferred Write

A performance option for use with UDBs to minimize I/O by deferring the writing of records.

DEFLOGON

The term used to describe an application entry path to be supported by SOLVE:Access. The DEFLOGON command is used to define application access paths and their associated text strings.

Dependent LU

Any logical unit that is made active by a command from the host system over a data link.

Dependent Processing Environment

An NCL processing environment which is a child process to another NCL process and hence has its output delivered as input to some other NCL procedure.

DOM (Delete Operator Message)

A non-roll deletable message (NRD) can be deleted from a window only when a DOM is issued.

Domain

1. An SNA term describing a domain that consists of the set of SNA resources controlled by one common control point called an SSCP. In terms of implementation, an SSCP is the host access method (VTAM). An SNA network consists of one or more domains.
2. A VTAM term that describes a logical division of a network. Networks are divided into domains that are associated with the way they are controlled.

Domain ID

A 1- to 4-character mnemonic used as a unique identifier for a system.

Dynamic Allocation

Assignment of datasets to a program at the time the program is executed rather than at the time the job is started.

EASINET

A component of SOLVE:Access. EASINET allows idle terminals to be brought under the control of SOLVE:Access and to be operated by installation-written NCL procedures that can provide a wide range of *front end* facilities to end-users of the network.

EDBS (External Database Support)

Facility allowing NCL to use DB2 databases.

EDS (Event Distribution Services)

A Management Services facility for notifying NCL procedures of events.

EIP (External Interface Package)

A facility that allows you to connect to, and issue commands on, a system from external sources such as TSO and BATCH jobs.

ER (Explicit Route)

The physical path between two network nodes. (SNA)

ESDS (Entry Sequenced Data Set)

A non-keyed VSAM dataset whose records are stored and retrieved in sequential order, and new records added to the end of the data set.

Exit

An installation-written routine that can be driven from a point within a program to provide data to the program, or perform additional processing relevant to that installation's specific requirements.

Extended Datastream

A 3270 datastream containing fields that utilize color and/or extended highlighting capabilities of the terminal.

External Database Connection

A term used to refer to connections from Management Services to the database which use the Call Attach Facility (CAF) to connect to DB2. NCL connections are connections to DB2 that utilize the Management Services connection. See also *Management Services Connection* and *NCL Connection*.

External Database ID

The ID of the external database as it is known to the operating system.

FSP

A Fujitsu operating system.

FTS (File Transmission Services)

A feature of Management Services which is used to transmit datasets between CPUs.

Function Key

A key on a terminal's keyboard which causes a panel to be completed. In the case of 3270, this term also applies to the ENTER key (the same behavior can be achieved on a 6530). Also called Program Function (or PF) keys.

ID

Identification

IMS (Information Management System)

IBM's database/data communication (DB/DC) system that provides a hierarchical database manager and transaction processing system.

Independent LU

A logical unit that does not receive an active LU over a link. Can act as a primary LU or secondary LU, and can have one or more LU-LU session at a time.

Initiator

The component of FTS which schedules transmission of a dataset.

INMC (Inter-Management Services Connection)

This facility allows systems running in a network to communicate with each other, providing general-purpose data transfer within the network.

INMC/EF (INMC/Extended Function)

INMC/EF provides the capability for up to sixteen sessions between any pair of systems. In appropriate systems, these sessions can traverse different physical network paths, thus increasing throughput. This component also provides additional link security and management facilities.

Internet
A wide area network connecting networks all over the world. Uses TCP/IP as the standard for information transmission.

I/O
Input/Output.

IS
Information System.

ISR (Inter-System Routing)
Provides centralized control at the system level through INMC.

KSDS (Key Sequenced Data Set)
A VSAM dataset whose records are directly accessed by a user-supplied key.

LAN (Local Area Network)
A computer network within a limited geographical area. Is not subject to external regulations.

Link
A term used to describe a logical connection between two peer communications systems such as two systems. See also *INMC (Inter-Management Services Connection)*.

LOGMODE
A VTAM term used to describe a set of characteristics and protocols of a logical unit.

Logon-path
A path through which users of SOLVE:Access gain access to other VTAM application programs. Paths are defined and controlled by the DEFLOGON command.

LOGPROC
The name given to an NCL procedure used to process messages destined for the Management Services activity log.

LSR (Local Shared Resources)
A technique for buffering I/O to VSAM files called LSR pools. NCL supports this type of processing for User Databases (UDBs).

LU (Logical Unit)
SNA introduced the concept of the logical unit (LU). The LU is a type of SNA network-addressable unit (NAU) that provides protocols for end users to gain access to the network and to the functional components of the LUs.

LU0

An unconstrained SNA protocol that allows implementers to select any set of available protocol rules, as long as the two LUs are able to communicate with each other successfully according to the rules chosen. Therefore, all LU types are an implementation of LU Type 0.

LU1

A line-by-line or typewriter type terminal (for example 3767, 3770), using SNA protocols.

LU2

A 3270 type terminal using SNA protocols.

LU3

LU Type 3 was implemented to support printers with a different data stream format. LU Type 3 is used by printers attached to an IBM display cluster controller.

LU4

LU Type 4 was implemented so that office system products could transfer documents.

LU6.2

A protocol that serves as a port into an SNA network. LU6.2 defines a specific set of services, protocols, and formats for communication between logical processors. LU6.2 provides presentation services for presentation of data to the end user, transaction services for performing transaction processing on behalf of the end user and LU services for managing the resources of the LU.

LU7

An SNA protocol that is used by word-processing devices.

MAI (Multiple Application Interface)

A facility of SOLVE:Access which is used to provide sessions with any number of other VTAM application programs from one terminal.

MAI/EF (MAI/Extended Function)

A SOLVE:Access facility which provides an extension to NCL to support session scripts which allow an NCL procedure to control the application and terminal session flow. MAI/EF also includes the Session Replay Facility (SRF) which allows recording and replay of session scenarios, and Screen Image Services (SIS) which allows recording of screen images for later retrieval or to send to another user.

MAI-FS (MAI Full Screen)

A SOLVE:Access facility that allows a single terminal to be used to provide full screen access to any number of other applications. The user may 'jump' from one application to another using designated 'jump' keys on the keyboard, or special command strings.

MAI-OC (MAI Operator Control)

A Management Services facility that allows an operator in OCS to have LU Type-1 sessions with many other applications from an OCS window. When used in conjunction with MSGPROC NCL procedures this can provide automated central monitoring and operation of multiple applications from the one operator console.

Management Services

This the central core of functions and service routines within the system. It supports all of the products.

Management Services Connection

A Management Services connection is a communication link from Management Services to the external database. It is started, stopped, defined, and deleted using the EDB command. A Management Services connection must be defined and started before opening an NCL connection. See also *NCL Connection*.

Mapping Services

A facility of Management Services that enables programmers to define complex data structures for use by applications.

MDS-MU (Multiple Domain Support-Message Unit)

The message unit used in data transmission between management applications in SNA networks.

MDO (Mapped Data Object)

Any data item that can be represented as a continuous string of bytes in storage.

Message Partition

A term associated with network partitioning that describes the group of network resources for which a user ID will receive unsolicited (PPO) VTAM messages.

MIB (Management Information Database)

Any database that provides information about the structure and capabilities of a management application.

Modify Interface

A means of communicating with an application program from the system console in OS/VS systems. The MODIFY command, abbreviated to F, avoids having an outstanding REPLY at the system console. Management Services supports the use of the MODIFY command.

MSGPROC

An NCL procedure used to intercept and process messages destined for a user's Operator Console Services (OCS) window.

MSP

An operating system for large scale Fujitsu systems.

MVS (Multiple Virtual Storage)

An IBM operating system. Fujitsu's functional equivalent is MSP.

NAU (Network Addressable Unit)

In SNA, a logical unit, a physical unit, or a system services control point. The NAU is the origin or destination of information transmitted by the path control network.

NCL (Network Control Language)

The interpretive language that allows logical procedures (programs) to be developed externally to Management Services and then executed by Management Services on command. NCL contains a wide range of logic, built-in functions and arithmetic facilities which can be used to provide powerful monitoring and automatic control functions.

NCL Connection

An NCL connection is a communication link from an NCL procedure to the external database. An NCL connection is opened and closed with the &EDB verb. A Management Services connection must be defined and started before opening an NCL connection. See also *Management Services Connection*.

NCL Procedure

A member of the procedures dataset comprising NCL statements and Man commands. The NCL statements and other commands are executed from an EXEC or START command specifying the name of the procedure.

NCL Process

The NCL task that is invoked, usually by a START command to execute one or more associated procedures. Each NCL process has a unique NCL process identifier.

NCL Processing Environment

Provides the internal services and facilities required to execute NCL processes for the user, from its associated window.

NCL Processing Region

All users (real or virtual) have an NCL Processing Region associated with their user ID while logged on. This region provides all of the internal services needed to allow the user to have processes executed on their behalf.

NCLID

A 6-digit NCL process identifier that is unique within the system. It is used to identify a process for the purpose of communicating with that process.

NCP (Network Control Program)
This resides within and controls the operation of a communications controller. The NCP communicates with VTAM.

NCPView
A component of NetMaster for SNA that allows monitoring of NCP configuration.

NCS (Network Control Services)
A facility of NetMaster for SNA that allows display and control of SNA network resources.

NDB
NDBs are field oriented databases in which data is stored and retrieved in named fields. They are designed for environments that have complex, high volume data storage and retrieval requirements.

NEWS (Network Error Warning System)
A facility of NetMaster for SNA which is used to provide network error and traffic statistics and error alert messages.

NMINIT
The NCL procedure automatically executed after system initialization has completed. It cannot contain commands that require VTAM facilities as it is executed before the primary ACB is opened. The procedure name can be changed by the installation.

NMREADY
The NCL procedure automatically executed once the primary ACB is open. It can contain commands that require VTAM facilities. The procedure name can be changed by the installation.

NMVT (Network Management Vector Transport)
A request/response unit (RU) that flows over an active session between a physical unit (PU) and a control point (CP).

Node
A connection point in a communications network.

NPF (Network Partitioning Facility)
A facility of Management Services that allows the range of resources which an operator can influence to be denied.

NPF Control Member
A member of the NPF dataset which defines a list of member names that are to be the resource tables for the associated user ID.

NPF Resource Table
A member of the NPF dataset that defines a group of network resource names. The resource names can be defined specifically or generically using wildcard characters. A resource table is addressed via a control member.

NRD (Non-Roll Delete) Message
A message that will not roll off an OCS window display until explicitly deleted. See *DOM*.

NT 2.1
Node Type 2.1. A node in an SNA network. It implements a peer-to-peer protocol and allows greater dynamics in network configuration, greater independence in session set up between partner LUs and reduced definitions. Same as PU type 2.1.

NTS (Network Tracking System)
A facility of NetMaster for SNA used to provide SNA session monitoring, dynamic online network tracing, accounting, and response time information in conjunction with diagrammatic representations of session partners.

Object Services
An object-oriented development environment used to define and maintain the application data and methods of some of the products.

OCS (Operator Console Services)
A facility of Management Services that provides general operational control and an advanced operator interface to VTAM for network management.

OSI (Open Systems Interconnection)
A set of ISO standards for communication between computer systems.

Packet
The unit of data used in transmission.

Panel Maintenance
A facility of Management Services which allows users to generate and modify panel definitions used for presentation purposes by NCL procedures. In releases prior to Version 3.0, this function was known as Edit Services.

Panel Services
A facility of Management Services for displaying full-screen panel definitions.

Panel Skip
The ability to chain menu selection requests together without having to display intermediate selection panels.

Password

A 1- to 8-character string chosen by a user and linked to their user ID for security purposes. To gain access to the system a user must enter both their defined user ID and its associated password.

PDS (Partitioned DataSet)

A type of dataset format that supports named data segments in the one physical dataset.

PFK (Program Function Key)

See *Function Key.*

PIU (Path Information Unit)

An SNA packet.

PLU (Primary Logical Unit)

Relates to SNA. A type of LU that is usually used by the application programs in a host. It refers to the BIND sender for a session. See also *Primary and Secondary.*

PPI (Program-to-Program Interface)

PPI is a general-purpose facility which allows programs, written in any language, to exchange data.

PPO (Primary Program Operator)

A VTAM term that describes a facility of VTAM that allows unsolicited network messages to be delivered to an application program, such as Management Services, for processing.

PPOPROC

The name given to the NCL procedure used to process unsolicited VTAM (PPO) messages.

Preload

A term applied to NCL procedures which are loaded into the system before being required, to improve system performance.

Primary Menu

The first menu of an application.

Primary and Secondary

Primary and secondary are SNA terms for describing the LU's role when the session is established. The primary LU sends the BIND request that causes the session to be established, and the secondary LU receives the BIND request. Rules defined in the BIND request determine which of these is the first speaker in the exchange of information.

PSM (Print Services Manager)

PSM is a facility of Management Services which simplifies the control of the physical printing of reports on JES or network printers.

PU (Physical Unit)

The part of a control unit or cluster controller which fulfils the role of an SNA-defined physical unit. Each node (a logical grouping of hardware) in an SNA network is addressed by its PU. There are 4 types of nodes or PU in an SNA network: PU-T5, PU-T4, PU-T2, PU-T1. A PU is a type of NAU. (See *NAU (Network Addressable Unit)*.

PU Type 1

A type of Physical Unit or Node in an SNA network. Consists of a terminal (such as an IBM 3278).

PU Type 2

A type of Physical Unit or Node in an SNA network. Consists of a cluster controller (such as an IBM3274, 3276, 3770 or 3790).

PU Type 4

A type of Physical Unit or Node in an SNA network. Consists of a communications controller (such as an IBM 3704, 3705, 3725 or 3745).

PU Type 5

A type of Physical Unit or Node in an SNA network. Consists of a host computer system (such as an IBM30*xx*, S/370 or 43*xx*, running VTAM).

PVC

X.25 Permanent Virtual Circuit.

Report Writer

A facility of Management Services which allows the creating and customizing of report definitions.

Request Unit (RU)

A message unit in an SNA network that contains control information such as a request code, or function management headers, end-user data, or both.

Reserved word

The term given to a token that will terminate an input expression if found unquoted outside the topmost parenthesis level. In the context of a particular verb statement the verb keywords are reserved words. A reserved word has special meaning for the current statement only, and different statements have different reserved words.

Resource Table

A term associated with NPF that describes a list of resource names or generic resource names that define a command or message partition.

Response Unit (RU)

A message unit in an SNA network that acknowledges a request unit.

Return Code

A code returned from the system that indicates the success or failure of the task performed.

ROF (Remote Operator Facility)

A facility of Management Services that allows an operator to sign on to a remote location, execute commands and have the results returned.

RSM (Resource Status Monitor)

A generic facility for monitoring and controlling and resources (not only SNA resources) for which configuration information is available.

RTM (Response Time Monitor)

A facility provided by IBM's 3*x*74 control units to monitor end-user response times. NEWS can interpret this data .

SAW (Session Awareness Data)

Network management data supplied by VTAM and processed by NTS.

SDLC (Synchronous Data Link Control)

A discipline for managing information transfer over a communications link.

Sequence Number

A number assigned to each message exchanged between a VTAM application program and a logical unit. Values increase by one throughout the session, unless reset by the application program using an STSN or CLEAR command.

Server

A process designed to serve the data to a client, or request process, for one or more users.

Session Name

A name assigned to a workstation or session to permit it to receive messages or share resources.

SIS (Screen Image Services)

A part of the MAI/EF facility of SOLVE:Access which provides the ability to record screen images for later retrieval or to send to another user.

SIS (Sequential Insert Strategy)

A technique that governs the insertion rule for data in VSAM KSDS. Normally used when ascending keys are being added to a dataset, to help in efficient use of space within the dataset.

SLU (Secondary Logical Unit)

(SNA) A type of LU that is usually used by the end-users at the terminals or by programs which reside in the peripheral node. See also *Primary and Secondary.*

SMF (System Management Facility)
An optional control feature of OS/VS that provides the means for gathering and recording information that can be used to evaluate system usage.

SMP (System Modification Program)
An IBM program used to install software on OS/VS1 and OS/VS2 systems.

SNA (Systems Network Architecture)
This term describes the logical structure, formats, protocols, and operational sequences for transmitting communication data through the communication system (Fujitsu equivalent is FNA). A set of standards that allows the integration of all the different IBM hardware/software products into a universal network. Introduced in 1974.

SNI (SNA Network Interconnection)
The connection of independent SNA networks using gateways.

SNMP (Simple Network Management Protocol)
An Internet standard for network management.

SPO (Secondary Primary Operator)
A VTAM facility that allows solicited network messages to be delivered to an application program.

SRF (Session Replay Facility)
A part of the MAI/EF facility of SOLVE:Access which provides the ability to record and playback terminal session scenarios.

SSCP (System Services Control Point)
A part of VTAM and the focal point of SNA networks. It controls general management of each domain.

Structured field
Representation of user ID attribute information exchanged between Management Services and its security exit.

Subtask
A unit of work that is established by a main task and is displaceable by the operating system.

SVC
X.25 Switched Virtual Circuit.

SYSPARMS
System parameters—values that affect system capabilities. Most SYSPARMS can be modified dynamically.

TCP/IP (Transmission Control Protocol/Internet Protocol)
A set of communications protocols that support peer-to-peer connectivity functions for both LANs and WANs.

TSO (Time Sharing Option)

An IBM product that allows terminal operators to interact directly with computer resources and facilities. Used mainly by application and system programmers. (Fujitsu equivalent is TSS).

UAMS (Userid Access Maintenance Sub-system)

The security component of Management Services that supports the definition of authorized users and their associated function and privilege levels.

UCS (Universal Character Set)

A printer feature that permits the use of a variety of character sets.

UDB (User Data Base)

1. UDB file access method layer allowing file access from NCL.
2. A term used to identify VSAM datasets to which NCL procedures may have access using the &FILE verb (GET, PUT, ADD, and DEL options).

User ID

Defines the function and privilege level to which a specific user is entitled when they sign on to the system. It is associated with a secret password to prevent use by unauthorized personnel. This definition is stored in the UAMS dataset or on an external security system.

User Services

A facility of Management Services that allows the creation of a range of specific procedures for different users, or classes of user, if required.

USS (Unformatted System Services)

A VTAM term that describes a facility that translates an unformatted command such as LOGON or LOGOFF, into a field formatted command for processing by formatted system services. Applies to terminals before connection to an application.

Verb

The term given to a stand-alone statement in an NCL program. NCL *verbs* cause actions to occur. There are different types of verbs, some that dictate the flow of processing and logic, others that fetch information for the procedure to process and others that cause data to flow to external targets.

VFS (Virtual File Services)

The VSAM dataset, used by many facilities as a database.

VM (Virtual Machine)

A superset operating system that allows other operating systems to run as if they each had their own machine.

VOS3

A Hitachi operating system comparable to IBM's MVS.

VR

(Session) Virtual Route.

VSAM (Virtual Storage Access Method)

A method for processing data files that utilizes relative, sequential, and addressed access techniques.

VSE (Virtual Storage Extended)

An operating system for small IBM systems.

VTAM (Virtual Telecommunications Access Method)

A suite of programs that control communication between terminals and application programs.

WAN (Wide Area Network)

A network that provides communication services to a large area, for example, a telephone network.

Wildcard

The term used to describe the character used (usually an asterisk) when defining resources generically—no specific matching character is required in the wildcard character position.

X.25

An international recommendation regarding the connection of computer equipment to public data networks.

XNF

A Hitachi network access method for OSI networks.

XSP

A Fujitsu operating system. Successor to FSP.

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W